

Programação em C++ utilizando o Framework Qt

Material de apoio às aulas

Este documento consiste em um material de apoio às aulas online do **Curso Programação em C++**. Pode ser utilizado como guia de acompanhamento e memória para anotações de conceitos desenvolvidos durante o curso, com o objetivo de facilitar o aprendizado e permitir a memorização do conteúdo. Este material deve ser utilizado em conjunto com a aula interativa, como parte integrante do curso e, em nenhuma hipótese, substitui o conteúdo online. É proibida a divulgação, comercialização e reprodução total ou parcial deste conteúdo.

Apresentação

- Introdução
- Lógica de Programação
- Estruturas Básicas do C++
- Orientação a Objetos com C++
- Interface Gráfica Usando Qt

Introdução

- C++ Histórico e Aplicações
- C++ Características
- Qt Framework

Anotações

Aula 1 - Introdução a Programação

1- C++ Históricos e Aplicações

- 30 anos
- Evolução do C++
- Novos paradigmas
- Forte mercado
- Aplicações conhecidas

A linguagem C++ foi concebida no final da década de 70 e início dos anos 80 como uma evolução da linguagem C, que na época já possuía uma década de existência bem sucedida.

Tal evolução, quase natural, se deu para incorporar novidades nas metodologias de desenvolvimento de software que culminaram no conhecido paradigma da orientação a objetos, cujo objetivo foi permitir a construção de softwares complexos de maneira mais organizada.

Dez anos depois de sua criação, a linguagem C++ já ocupava um posto importante no mercado internacional de produção de software, se mantendo até hoje como líder em algumas áreas, principalmente no desenvolvimento de software para o mercado desktop.

Anotações

Tal mercado consolidado pela linguagem C++ no mundo desktop inclui o próprio sistema operacional Windows, em todas as suas versões. Também está incluso o pacote Office e os navegadores dominantes no mercado: Chrome, Firefox, IE e parte do Safari. É curioso lembrar também que a máquina virtual Java da Sun também é em boa parte feita em C++ por questões de desempenho.

2 - C++ Características

- Portável: Diversidade de ambientes
- Compilada: Gera executáveis nativos
- Não-gerenciada: Acesso total a memória
- Procedural, funcional e orientada a objetos

Do ponto de vista teórico, todas as linguagens de programação são equivalentes, porém na prática saber entender as características das dezenas de linguagens de programação importantes que estão presentes no mercado nos ajuda a decidir quando devemos aprender e usar determinada linguagem.

O primeiro aspecto importante é que C++ é uma linguagem portável. Isto significa que nos seus 30 anos de existência, houve uma grande esforço por parte dos fabricantes e da comunidade de desenvolvimento de software em criar compiladores e bibliotecas de sistema para que código produzido em C++ possa rodar nos mais diversos tipos de processadores e de sistemas operacionais.

Anotações

Sendo possível utilizar C++ para escrever software para dispositivos diversos, desde minúsculos sensores até poderosos supercomputadores.

Em segundo lugar, é importante saber que programas feitos em C++ são tipicamente compilados (transformados) em código nativo do processador, sendo executado com o maior desempenho possível e acesso livre a memória em um ambiente não-gerenciado, onde o programador tem total controle sobre a execução do programa. A contrapartida disso tudo é que com a liberdade vem junto a responsabilidade de utilizar e gerenciar de forma correta a memória disponível.

Do ponto de vista técnico é importante saber que a linguagem C++ comporta tanto os paradigmas: procedural, funcional e orientado a objetos, sendo este último o foco deste curso.

3 - *Qt Framework*

- Conjunto de bibliotecas
- Início: Apenas interface gráfica
- Compatível: OSX, Windows, Linux, Android,
- Licenciamento Comercial e Aberto
- Ferramentas amigáveis: Qt Creator

Anotações



A linguagem de programação C++, sozinha, não é suficiente para desenvolver software complexos, visto que o núcleo da linguagem tem o compromisso de ser mínimo e não possuir nenhuma dependencia ou relação com sistemas operacionais. Desta forma para se desenvolver software “de verdade” é necessário o auxílio de bibliotecas.

Cada sistema operacional (Windows, Linux, Android, OSX) possui diversos agrupamentos de bibliotecas diferentes para utilização junto a linguagem C++. Todo esse ecossistema disponível, na prática, deixa o iniciante muito confuso e acaba servindo como uma barreira para a utilização do C++.

Com isso em mente, o framework Qt (pronuncia cute) foi desenvolvido como um amplo conjunto de bibliotecas para a linguagem C++, tentando cobrir todas as necessidades inerentes a um programa, como: Exibição de componentes gráficos (Telas), acesso a arquivos, banco de dados, comunicação de rede, reprodução de som, vídeo e imagem. Sendo que todas as funcionalidades do framework são disponíveis para utilização em diversos sistemas operacionais, sem que o programador precise alterar o código de seu programa para poder compila-lo para plataformas diferentes.

Somadas todas as funcionalidades das bibliotecas que compõe o Qt, com seu licenciamento como software-livre, seu suporte a múltiplas plataformas e uma plataforma integrada de desenvolvimento que veremos a frente, o Qt Framework é uma escolha excelente para o desenvolvimento de aplicações C++ para desktop.

Anotações

4 Apresentação do Qt Creator

4.1 Instalação

4.2 Organização dos Painéis

4.3 Criação de um novo projeto

4.4 Compilação e execução

4.4.1 Problemas na compilação (permissões e locais)

Anotações

4.5 Depuração e pontos de parada

5 Lógica de Programação

- Organização de um Programa
- Variáveis e Operações
- Funções: Parâmetros e Retorno
- Controle de Fluxo e Lógica Booleana
- Controle de Repetições

Um programa de computador visto de uma maneira estritamente simplificada, nada mais é do que uma sequência de instruções as quais um processador é capaz de executar. Tais instruções são capazes de lidar com o armazenamento e a manipulação da informação de uma maneira numérica. Da mesma maneira que as instruções alteram a informação presentes no computador, o valor destas informações também pode ser usado para controlar o fluxo da execução do próprio programa, dando espaço as infinitas possibilidades de softwares que se pode executar.

Desta maneira, um programa em execução pode ser visto como um conjunto de informações armazenadas somadas a um conjunto de instruções sendo processadas.

Anotações

Para facilitar o desenvolvimento de programas, sem a necessidade de o programador conhecer diretamente as instruções que um processador é capaz de executar, foram criadas as chamadas linguagens de alto nível (que se aproximam da linguagem natural humana), entre elas o C++ que veremos neste curso.

Antes de entrar nos detalhes da linguagem C++ podemos aprender os conceitos básicos de programação usando uma linguagem informal, próxima a linguagem falada, para simular a execução de um programa.

5.1 Organização de um programa

- Programa: Conjunto de Dados e Instruções
 - Código fonte ⇒ Arquivos de Texto
 - Divisão em Componentes
 - Variáveis ⇒ Dados
 - Funções ⇒ Comportamento (instruções)

Escrever um grande programa de computador é uma tarefa complicada que exige um alto grau de conhecimento e organização. Num primeiro momento, o código-fonte, ou seja, o texto que constitui o que o programa de computador deve fazer é só isto, apenas um texto.

Anotações

Para facilitar a organização do programa, seu código fonte deve ser quebrado em vários arquivos. Cada um desses arquivos pode ser visto como um pequeno componente que combinado dará origem ao programa por completo.

De maneira simplificada, escreveremos programas constituídos por funções e variáveis.

Dentro das variáveis serão armazenados e manipulados os dados do programa. As funções por sua vez são responsáveis pela lógica, ou comportamento, do programa que pretendemos construir.

5.2 Variáveis e operações

- Possuem um nome
- Tipo determinado
- Escopo
- Valor

Anotações

```
Numero Inteiro Contador = 10
Numero Real Distancia = 105.10
Texto Mensagem = "ABCDEFG"
```

- Dados numéricos (Inteiros e Reais)
- Operações numéricas: + - / *

```
Numero Inteiro Contador = 10
Numero Inteiro Incremento = 2

Contador = Contador +
Incremento
```

- Dados numéricos (Inteiros e Reais)
- Operações numéricas: + - / *
- Ordem de execução (Precedência)

Anotações _____

```
Numero Inteiro Contador = 10  
Teste = (Contador + 1) * 2
```

- Tipo Booleano (Verdadeiro e Falso)
- Tipo textual (Texto e Caracteres)

```
Booleano Condicao = Falso  
Texto Mensagem = “Olá Mundo”  
Mensagem = Mensagem + “!”  
Caractere Letra = ‘a’
```

Uma variável é um nome que o programador escolhe para guardar determinado valor. Toda variável possui um tipo, que designa qual tipo de valor a variável irá armazenar. Além disso, toda variável possui um escopo. Sendo que o escopo é o contexto onde a variável é especificada pelo programador. Em um determinado ponto do código-fonte, o programador só pode utilizar variáveis criadas no escopo em que o ponto se encontra ou em algum escopo “pai” deste.

Anotações

De maneira resumida, toda variável possui um nome, um tipo, um escopo e um valor. Sendo que todos esses atributos são determinados pelo programador. Depois que uma variável é criada, a única coisa que se pode mudar é seu valor. A quantidade de variáveis que se pode criar é em tese infinita, e na prática restrita a capacidade do compilador e da quantidade de memória disponível para execução do programa.

Embora os valores armazenados em uma variável internamente sejam apenas números, utilizamos durante o desenvolvimento tipos de dados mais elaborados, sendo os principais tipos números inteiros, números reais, valores booleanos (verdadeiro e falso) e sequências de caracteres (textos).

Utilizando variáveis numéricas, podem aplicar operações aritméticas básicas, como a soma (+), a subtração (-), a divisão (/) e a multiplicação (*). Outras operações mais complexas como logaritmos e funções trigonométricas são fornecidas por bibliotecas especiais.

Exibir declaração de variáveis e operações aritméticas

O tipo booleano comporta o armazenamento de valores Verdadeiro e Falso e a aplicação de operações lógicas como a conjunção, a disjunção e a negação. Tais operações serão detalhadas no futuro quando virmos controle de fluxo.

Variáveis textuais são sequências de tamanho variável cujo texto pode ser acessado de maneira livre e manipulado em operações como concatenação e extração.

Para fins didáticos, não é necessária nenhuma formalidade ao utilizar as operações para escrever pseudo-códigos. O importante nesta etapa é expressar idéias e não a corretude sintática dos comandos.

Anotações

5.3 Funções: Parâmetro e retorno

- Comportamento do Programa
- Ponto de Entrada
- Valores de Entrada \Rightarrow Parâmetros
- Valor de Saída (Opcional)

```
Funcao Soma(Inteiro A, Inteiro B, Inteiro C)
    Retorna A + B + C
    Fim
    ...
    Inteiro Resultado = Soma( 1, 2, 3)
```

As funções são a alma do nosso programa. Nestas definimos quais comandos e operações deverão ser executadas. Cada função deve ser escrita de maneira a realizar uma tarefa específica. Toda função possui as seguintes propriedades: um nome, um tipo de retorno, uma lista de parâmetros e o corpo da função.

Anotações

Todo programa possui uma função especial de “inicio”, que é a primeira função executada do programa. A partir desta é possível chamar as demais funções.

O conjunto de propriedades nome, tipo de retorno e lista de parâmetros é chamado de assinatura da função, pois identifica a mesma.

O nome da uma função identifica a mesma para ser usada em outros pontos do programa e deve ser escolhido de maneira a identificar claramente qual o objetivo ou comportamento esperado da função.

Toda função possui valores de entrada e valor de retorno, também sendo possível também a existência de funções sem valor de retorno. Os valores de entrada são fornecidos através dos parâmetros da função. Cada parâmetro é uma variável dentro do escopo do corpo da função cujo valor é determinado no momento em que a função é chamada. Sendo uma variável, cada parâmetro possui um tipo que determina o que é que pode ser passado como parâmetro a este.

O valor de saída de uma função tem seu tipo determinado na assinatura da função e seu conteúdo é determinado pelo código contido no corpo da função.

Anotações

5.4 Controles de Fluxo e Lógica booleana

- Execução condicional
 - Condição Usa Operadores Booleanos e Comparativos
 - SE-SENÃO

```
Funcao Positivo ( Inteiro Numero )  
    SE Número >= 0  
        Retorna Verdadeiro  
    SENÃO  
        Retorna Falso  
Fim
```

- Comparativos
 - Igual: ==
 - Diferente: !=
 - Menor que: <
 - Maior que: >
 - Menor ou igual: <=

Anotações

- Maior ou igual: \geq

```
Funcao Par ( Inteiro Numero )  
  SE Resto( Numero, 2 ) == 0  
    Retorna Verdadeiro  
  SENÃO  
    Retorna Falso  
Fim
```

- Operadores Lógicos
- Negação: NÃO
- Conjunção: E
- Disjunção: OU

Anotações _____

```
Funcao Positivo ( Inteiro Numero )
```

```
  SE Número >= 0
    Retorna Verdadeiro
  SENÃO
    Retorna Falso
```

```
Fim
```

O controle de fluxo é um dos pontos chaves que faz do computador uma ferramenta tão poderosa. Do ponto de vista do programador, o controle de fluxo é feito através de funções de comparação que determinam a execução condicional de determinados pontos do código.

O principal componente do controle de fluxo é o par: SE - SENÃO, do inglês IF-ELSE.

Na estrutura SE-SENÃO utilizamos variáveis do tipo booleano ou operações cujo resultado é um tipo booleano, ou seja, Verdadeiro ou Falso.

Exibir exemplo aqui

Para dar poder ao controle de fluxo, podemos usar operações de comparação como:

- IGUAL
- DIFERENTE
- MAIOR QUE / MENOR QUE

Além disso, podemos comparar diversos resultados de comparações usando os operadores lógicos E, OU e NÃO.

Anotações

5.5 Controle de Repetições

- Execução de Código Repetida por Uma Condição
- ENQUANTO

```
Enquanto Numero > 0
    Resultado = Resultado * 2
    Numero = Numero - 1
Fim
```

- Execução de Código Repetida em um Intervalo
- PARA-FAÇA

Anotações

```
Para Inteiro i de 0 até Numero Faça  
    Resultado = Resultado * 2  
Fim
```

O controle de repetições é uma continuidade do controle de fluxo onde é possível determinar blocos de código cuja execução poderá se repetida sob determinada condição. A união do controle de fluxo e das repetição é o cerne do desenvolvimento da parte algorítmica dos programas, ou seja, a parte que lida com a manipulação da informação.

No controle de repetições, as principais funções usadas são o ENQUANTO e o PARA.

A função ENQUANTO executa um bloco de código enquanto uma dada condição for verdadeira. Da mesma maneira, a função PARA executa um bloco de código repetidas vezes enquanto manipula o valor de uma variável de um número inicial até um limite final.

As operações utilizadas no controle de repetição são exatamente as mesmas usadas no controle de fluxo.

Anotações

5.6 Estruturas Seqüenciais

- Múltiplos dados na mesma variável
- Todos do mesmo tipo
- Acessados através da posição

```
Vetor de Inteiros Numeros (Tamanho  
10)  
  
Numeros 1 = 100  
Numeros 2 = Numeros 1 * 2  
Numeros 3 = Numeros 2 * 2
```

Anotações

5.7 Exemplo

5.7.1 Parte 1: Leitura e armazenamento das notas

5.7.2 Parte 2: Calculo da média e exibição dos aprovados

6 Finalização

Anotações

Aula 2 Estruturas Básicas da Linguagem C+ +

Parte 1

Introdução

Iniciaremos agora o aprendizado da linguagem C++, começando pelos seus aspectos fundamentais. Nesta primeira unidade veremos como um programa em C++ é organizado e aprenderemos um subconjunto básico da linguagem C++ que se assemelha também a linguagem C, sua predecessora.

O conteúdo das próximas etapas está organizado da seguinte maneira:

1. Primeiros passos: Variáveis, Funções e Operações
 2. Laços condicionais, estruturas sequenciais e variáveis globais
 3. Memória dinâmica, ponteiros e referências
 4. Estruturas de dados e acessos a arquivos

Anotações

1 Primeiros passos: Variáveis, Funções e Operações

1. Organização e separação do código fonte
2. Variáveis e tipos primitivos
3. Operações aritméticas e precedência
4. Operações lógicas e de comparação
5. Funções
6. Controle de fluxo: IF-ELSE
7. Controle de fluxo: SWITCH-CASE
8. Escrita no console usando STL
9. Leitura do console usando STL

Anotações

1.1 Organização e separação do código fonte

- Arquivos de código fonte:

- Dividir o programa em unidades lógicas, ou seja, arquivos cujo conteúdo pertence ao mesmo componente, e;
- Separar o programa em unidades de compilação distintas, que agilizam o processo de recompilação após uma modificação no código.

- Arquivos de código fonte (2 tipos):

- Os HEADERS, ou seja, arquivos de cabeçalhos, cuja a extensão tipicamente é a (.h)
- E os SOURCES, ou seja, arquivos de código, com a extensão normal sendo (.cpp), embora também seja comum a extensão (.cc) e (.cxx)

Cada programa feito na linguagem C++ é dividido em diversos arquivos de texto. Sendo que esta divisão possui dois objetivos:

- Dividir o programa em unidades lógicas, ou seja, arquivos cujo conteúdo pertence ao mesmo componente, e;
- Separar o programa em unidades de compilação distintas, que agilizam o processo de recompilação após uma modificação no código.

Além disso, o código dos nossos programas deverão ser separados em dois tipos diferentes de arquivos:

Anotações

- Os HEADERS, ou seja, arquivos de cabeçalhos, cuja a extensão tipicamente é a (.h)
- E os SOURCES, ou seja, arquivos de código, com a extensão normal sendo (.cpp), embora também seja comum a extensão (.cc) e (.cxx).

Arquivos adicionais também estarão presentes no projeto, como os arquivo .pro, o Makefile e os arquivos intermediários como os arquivos com extensão (.o). Tais arquivos não são modificados diretamente pelo programador mas sim pelas ferramentas de desenvolvimento.

1. Arquivos de cabeçalho

Quando dividimos nosso programa em diversos arquivos de código-fonte diferentes, é necessário criar um arquivo de cabeçalho auxiliar a cada um dos arquivos de código fonte.

O objetivo destes arquivos de cabeçalho é possuir a assinatura, ou seja, a declaração das funções, variáveis globais e demais elementos que serão definidos no arquivo de código fonte.

Essa separação se tornará especialmente importante quando aprendermos orientação a objetos.

2. Arquivos de código fonte

Os arquivos de código fonte são o principal bloco na construção de um programa.

Nele devemos definir o corpo das funções que constituem nosso programa.

Vejamos agora este exemplo de um projeto com alguns arquivos de cabeçalho acompanhados de alguns arquivos de código fonte.

Anotações

Perceba que para usar um arquivo de cabeçalho em um arquivo de código fonte, utilizamos a diretiva INCLUDE.

Um detalhe final muito importante. Você pode incluir arquivos de cabeçalho em outros arquivos de cabeçalho, porém nunca deve incluir o arquivo de código fonte usando a diretiva INCLUDE em um outro arquivo de código fonte, pois isto irá gerar problemas na etapa de compilação ou na etapa de linkagem.

1.2 Variáveis e tipos primitivos

Numéricas	int, short, long, float, double
Lógica	bool
Caractere	char

Como vimos na unidade de lógica de programação, variáveis são locais onde podemos armazenar e manipular valores. Para isto cada variável possui um nome, um escopo e um tipo de dado atribuído pelo programador.

Anotações

Vejamos os seguintes exemplos:

Em C++ para usarmos uma variável precisamos primeiramente fazer sua declaração, especificando seu tipo de dado e seu nome. No momento da declaração é opcional atribuir um valor inicial a variável.

Por enquanto, não se preocupe com o escopo. Aprenderemos sobre ele gradualmente no futuro, já que o escopo da variável depende de fatores externos a ela.

Tome cuidado, pois não devemos utilizar o valor da variável enquanto esta não tiver sido inicializada, do contrário o valor contido nela será indefinido. Apenas variáveis declaradas no escopo global serão inicializadas com o valor zero por padrão.

As variáveis mais simples em C++ são as de tipos primitivos. Os tipos primitivos mais importante são os numéricos (inteiros e de ponto flutuante), os lógicos (ou booleanos) e os caracteres.

No caso dos tipos numéricos, também podemos especificar se eles possuíram sinal ou se serão apenas positivos. Vejamos os principais tipos primitivos:

Repita e execute esse mesmo exemplo e faça alguns experimentos trocando o valor das variáveis, tente descobrir qual o maior e o menor número que você consegue armazenar em cada uma delas.

Anotações

1.3 Operações aritméticas e precedência

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

Soma e atribuição	$+=$	Subtração e atribuição	$-=$
Multiplicação e atribuição	$*=$	Divisão e atribuição	$/=$
Resto da divisão e atribuição	$%=$		

Anotações

Pré-Incremento	<code>++var</code>	Pré-Decremento	<code>--var</code>
Pós-Incremento	<code>var++</code>	Pós-Decremento	<code>var--</code>

Agora que sabemos como declarar uma variável e iniciar seu valor, veremos como podemos alterar o conteúdo delas. Ou seja, veremos quais operações e operadores podemos usar com nossas variáveis.

O primeiro operador que já usamos foi o operador de atribuição, representado pelo símbolo (`=`). Chamamos o operador de atribuição de um operador binário, ou seja, que trabalha com 2 elementos, o da esquerda e o da direita. No caso do operador de atribuição, o conteúdo (valor) à direita é copiado para a variável a esquerda.

Se encaixam como operadores binários também a soma, a subtração, divisão, multiplicação e o operador de resto da divisão. Ao utilizarmos tais operadores os operandos da esquerda e direita podem ser variáveis ou literais, ou seja, valores expressos diretamente no código-fonte.

Podemos combinar a atribuição junto com uma operação básica, alterando o valor de uma variável a partir de uma operação envolvendo essa mesma variável e uma outra variável ou valor literal. Vejamos agora os exemplos de atribuição combinada com operação:

Anotações _____

A linguagem C++ ainda possui algumas operações especiais, que são apenas um jeito mais simples de expressar operações muito comuns, as principais são as operações de incremento e decremento, divididas em dois tipos:

Recomendo agora que você repita e modifique livremente estes exemplos para se habituar com os operadores.

1.4 Operações lógicas e de comparação

Negação (NOT)	!	OU-Exclusivo	^^
E (AND)	&&	OU (OR)	

Igual	<code>==</code>	Diferente	<code>!=</code>
Menor	<code><</code>	Maior	<code>></code>
Menor ou igual	<code><=</code>	Maior ou igual	<code>>=</code>

Anotações

- Precedência (order das operações)

- Primeiro: Operações unárias (de um só operando), como a negação
- Depois: Operações de comparação
- Por último: Operações lógicas binárias (E, OU, OU-Exclusivo)

As variáveis do tipo booleano, que representam valores de verdadeiro ou falso, são de importância fundamental quando aprendermos controle de fluxo e de repetições. Para isso precisamos aprender primeiramente como usá-las.

C++ possui as três operações lógicas básicas e mais uma grande quantidade de operações de comparação.

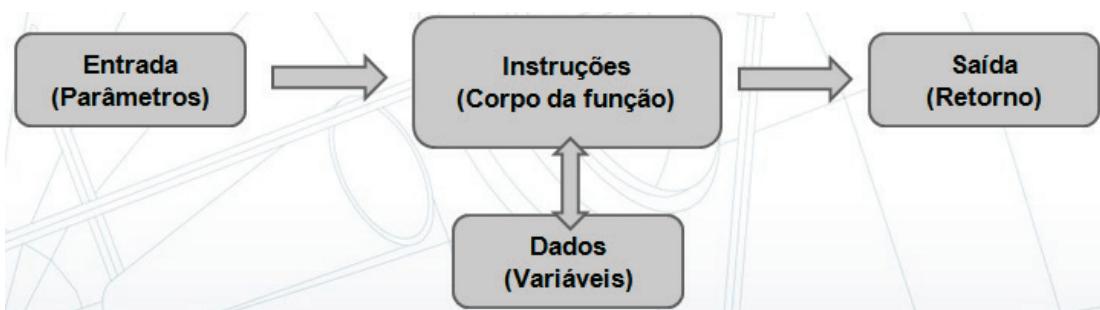
A precedência das operações lógicas e de comparação é a seguinte:

- - Primeiro são executadas as operações unárias, como a negação
- - Depois são executadas as operações binárias de comparação
- - E por último são executadas as operações binárias lógicas (E e OU)

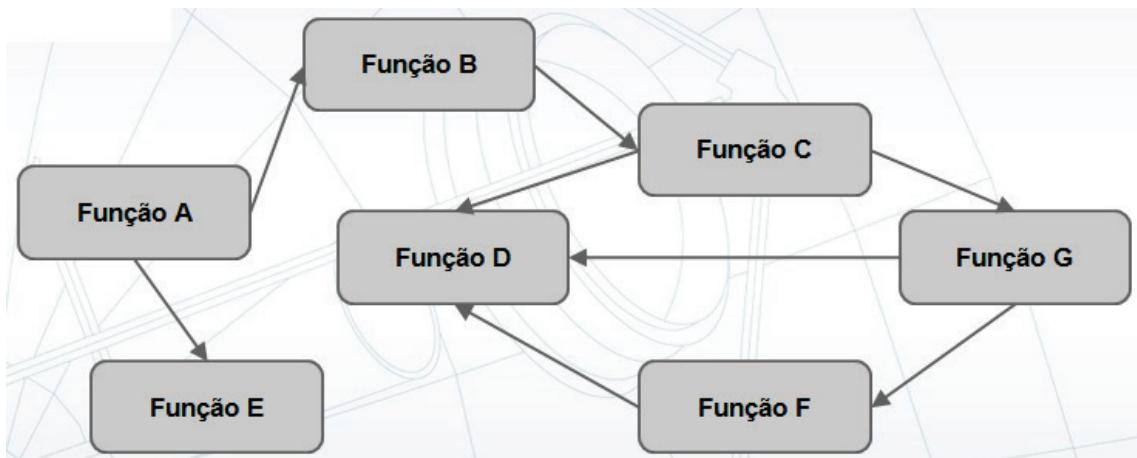
Quando misturar muitas operações lógicas e principalmente se foram feitas operações aritméticas junto, agrupe adequadamente todas as operações utilizando parênteses para garantir a ordem de avaliação correta de cada uma das expressões.

Anotações

1.5 Funções



Anotações



Chegamos agora num assunto fundamental, funções. O conceito de função numa linguagem de programação vem diretamente do conceito matemático. Ou seja, uma função é um conjunto limitado de instruções que manipulam um conjunto de valores de entrada para produzir um valor de saída, ou resultado.

Toda função possui uma assinatura, que é a descrição de seu tipo de entrada e saída acompanhada também do seu nome. Além disso toda função possui um corpo, delimitado pelos símbolos de abre e fecha chaves.

Quando dividimos nosso código em vários arquivos CPP, escrevemos apenas a assinatura da função no arquivo .H correspondente.

Vamos ver agora como funciona a declaração e utilização de um simples função.

Anotações

O objetivo de criamos funções é dividir o código em partes lógicas, com a possibilidade inclusive de utilizar a mesma função em vários pontos diferentes do programa. Tal funcionalidade se torna fundamental em programas extensos onde o mesmo tipo de operação precisa ser repetida em diversos pontos diferentes, podendo ser realizada pela mesma função.

Em C++ também temos um tipo especial de função, cujo retorno é do tipo VOID. Tal tipo serve para indicar que a função não retorna nenhum valor. Funções void também são chamadas de rotinas ou procedimentos.

A passagem de parâmetros para uma função deve ser feita na sua invocação, também conhecida como chamada. A ordem dos parâmetros é fundamental e deve respeitar a mesma ordem da declaração. Quando utilizamos uma variável como parâmetro, estamos copiando o valor desta variável para o parâmetro da função. Vamos analisar o próximo exemplo para entender melhor como isto funciona.

Alguns detalhes importantes:

Primeiro: Todo programa deve possuir uma função chamada main que é o ponto de entrada do programa. O valor de retorno e os parâmetros da função main não serão abordados por enquanto, logo você pode ignorá-los.

Segundo detalhe: O retorno da função pode ser feito em qualquer ponto do corpo da função, sendo que a execução das instruções dentro da função para exatamente no momento de retorno

Terceiro detalhe importante: Não podemos criar funções dentro de outras funções, apenas chamar funções que já foram declaradas.

Penúltimo ponto: A regra para nomear as funções é a mesma das variáveis, devemos usar apenas letras, números e underscore (traço baixo).

Anotações

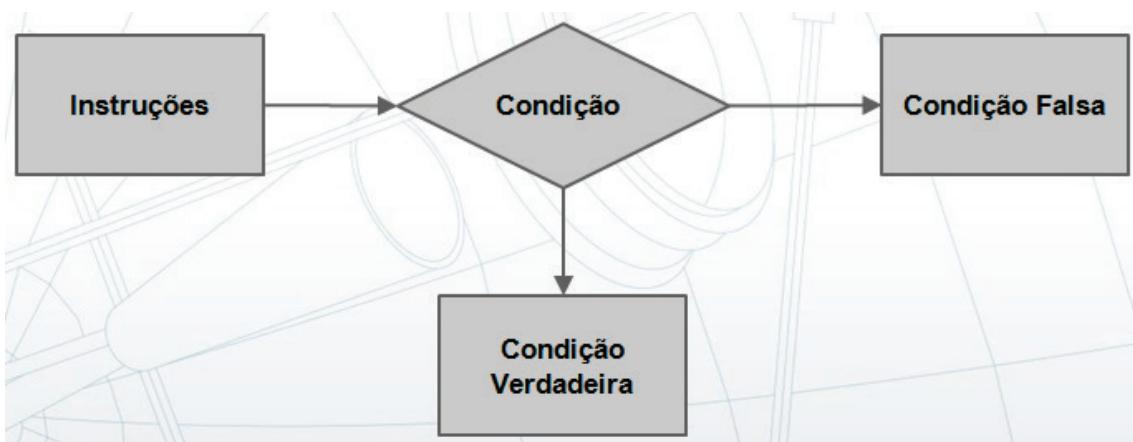
A linguagem C++ diferencia letras maiúsculas de minúsculas, logo devemos ficar atentos a isso. Também não podemos usar nomes de funções e operadores que já existam na linguagem e também não podemos declarar o nome da função ou variável iniciado com numeral.

E finalmente: Cada projeto ou cada equipe de desenvolvimento costuma adotar convenções na hora de nomear as funções, ao trabalhar em equipe adote o mesmo padrão dos demais colegas pois facilita bastante o desenvolvimento e a compreensão do código já existente.

Modifique o exemplo de função que você viu para realizar outras operações, como uma média, por exemplo. No futuro ainda veremos maneiras mais sofisticadas de criar e usar funções.

Anotações

1.6 Controle de Fluxo: IF-ELSE



Em lógica de programação vimos os comandos SE e SENÃO, utilizados para controlar o fluxo de execução do programa. Em C++, utilizamos os comandos IF e ELSE, que só podem ser utilizados dentro do corpo de funções.

O comando IF possui uma clausula, que quando verdadeira faz executar o código no corpo subsequente ao comando. Caso contrário, se existir um comando ELSE, o corpo após este será executado.

Anotações

É sempre uma boa prática criar um bloco delimitado por abre e fecha chaves após um IF ou ELSE, caso contrário apenas o primeiro comando subsequente ao IF é considerado como pertencente a ele.

Podemos usar livremente comandos IF dentro do corpo de outros comandos IF ou ELSE. Chamamos isto de IF aninhado.

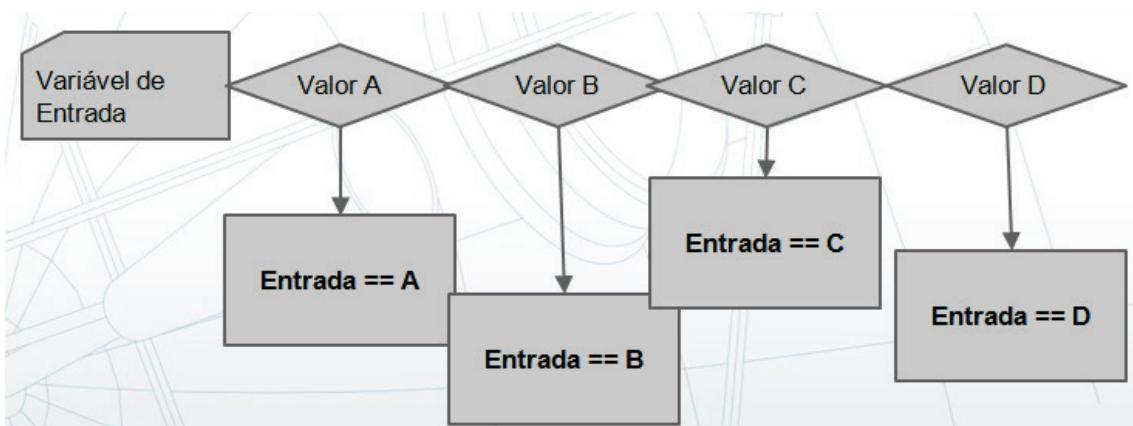
Um detalhe importante: Dentro do corpo dos comandos IF e ELSE podemos utilizar variáveis que foram declaradas no mesmo escopo que o próprio comando IF se encontra ou podemos declarar variáveis novas dentro do corpo do IF. Caso declaramos alguma variável nova, é preciso lembrar que o escopo dessa nova variável passa a ser o próprio corpo do IF e ela deixará de existir após o encerramento da execução do IF.

Mais uma dica: Fique atendo quanto ao operador de comparação (==). Um erro bastante comum é trocá-lo pelo operador de atribuição, cujo símbolo é apenas um caractere (=) e não dois.

E agora para finalizar. Existe ainda uma segunda forma de escrever expressões IF-ELSE. A chamada comparação em linha, que constitui uma operação ternária, ou seja, com 3 elementos. A comparação em linha permite escrever de maneira bem compacta expressões comuns usando geralmente apenas 1 linha no lugar de 3.

Anotações

7 Controle de fluxo: SWITCH-CASE



Em determinadas situações precisamos comparar uma variável numérica contra vários possíveis valores e, caso a combinação seja verdadeira, executar determinada série de comandos. Para este tipo de problema, podemos utilizar uma série de IFs ou então, de uma maneira mais eficiente e mais compacta, o comando SWITCH-CASE.

Na prática, o SWITCH-CASE funciona como um salto, um pulo no código usando o valor da variável de condição para passar a execução do código para o ponto onde o comando CASE possua o mesmo valor.

Anotações

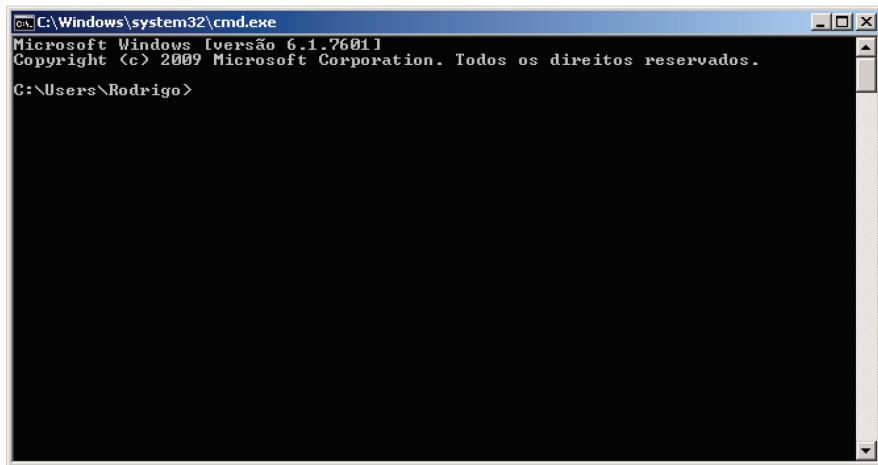
Uma diferença importante que deve ser notada no controle usando SWITCH-CASE, é que o fluxo é interrompido apenas pelo comando BREAK, ou seja, caso não seja utilizado, a execução continuara na próxima clausula CASE.

Outro ponto importante é que, caso nenhuma clausula CASE tenha o valor da condição SWITCH, é possível determinar uma clausula padrão, chamada default, que funcionará na prática como se fosse um ELSE no final de um sequência de IF-ELSE. Vamos voltar ao editor:

Um erro comum ao utilizar SWITCH-CASE é efetuar a declaração de novas variáveis dentro do corpo da clausula CASE. Tal declaração resulta em erros bem estranhos na hora de compilar o código. Para resolver o problema é necessário criar um bloco usando abre e fecha chaves dentro do case, ou seja, um novo escopo.

Anotações

1.8 Escrita no console usando STL



- STL = Standard Template Library
- Padrão C++
- Diversas funcionalidades básicas (Estruturas de dados, operações matemáticas, algoritmos, entrada e saída)

Quando escrevemos simples programas para teste e aprendizado, usando o terminal de texto, ou console, precisamos de ferramentas adequadas para fazer a entrada e saída de informações.

Anotações

Ou seja, para escrever na tela e ler caracteres digitados pelo usuário. Veremos agora o básico de escrita na tela e depois o básico de entrada de valores.

Para isso utilizaremos a biblioteca STL. Bibliotecas são componentes prontos que fornecem uma série de funções para realizar determinada tarefa. Neste caso, a STL (Standard Template Library) é a principal biblioteca padrão do C++ e fornece um enorme conjunto de funções e de estruturas sofisticadas. Para utilizar suas funcionalidades de entrada e saída precisamos incluir um cabeçalho, veja o exemplo:

Um detalhe importante que ainda não abordaremos a fundo. Todas as funcionalidades da STL estão em um namespace, ou seja, possuem um prefixo específico chamado STD. Para não precisarmos escrever STD antes de todos os elementos da STL, vamos utilizar o comando “using namespace std” para incorporar o namespace. Veja no exemplo:

Para escrever na tela, usaremos uma variável global chamada COUT e o operador de saída <<. Desta forma podemos encadear uma sequência de operações de saída de uma maneira fácil. Uma variável global chamada endl representa o caractere de quebra de linha que é usado com bastante frequência.

Quando escrevemos caracteres numéricos na saída, podemos usar alguns comandos adicionais da STL para controlar como o número será representado. Tal funcionalidade é útil quando desejamos mudar a base numérica, por exemplo de decimal para hexadecimal, ou alterar a quantidade de casas que desejamos exibir em números reais. Observe:

Para finalizar: Evite misturar o operador de saída com operações aritméticas, isto evita erros de compilação e melhora a legibilidade do código.

Anotações

1.9 Leitura do console usando STL

- Utilização muito semelhante a escrita
- Mesmo include de biblioteca: iostream
- Variável cin ao invés de cout
- Operador >> no lugar do operador <<

Ler dados a partir do console de texto é tão simples quanto escrever. Usaremos o mesmo include da STL, a iostream, porém dessa vez a variável global envolvida é CIN. Junto a ela usamos o operador >>, que tem o símbolo contrário ao operador de saída usado com COUT. O operador de entrada, deve ser acompanhado de uma variável, que será o destino do valor lido.

Confira este exemplo:

É possível consultar se houve algum erro na leitura através das operações cin.good() ou se não é mais possível ler usando cin.eof(). Tais operações são importantes quando criamos programas cuja entrada é o resultado da saída de outro programa. Tal operação costuma ser feita através de um PIPE e é muito comum na plataforma Linux.

Anotações

1.10 Exemplo 2

Vamos construir um pequeno programa para ler 2 números e 1 caractere representando cada uma das operações básicas. O objetivo do programa é exibir o resultado da operação, como se fosse uma calculadora bem primitiva.

- Primeira versão: Usando IF para tomada de decisão
- Segunda versão: Usando SWITCH para decisão

Anotações

Aula 3 - Estruturas Básicas da linguagem C++ Parte 2

1 Laços Condicionais, Estruturas seqüenciais e variáveis globais

A segunda unidade desta nossa aula tratará de três componentes importantes no desenvolvimento de software:

1. Laços condicionais, ou seja as repetições
 2. Estruturas sequências, para o armazenamento de séries de valores do mesmo tipo e por último:
 3. Variáveis globais, ou seja, variáveis compartilhadas e acessíveis em todos os lugares de um programa

Unidos os conhecimentos aprendidos na unidade anterior e nesta unidade, teremos visto todas as estruturas básicas já apresentadas na aula de lógica de programação. Com isso, é possível estudar e desenvolver uma série de pequenos programas e algoritmos. Vamos lá!

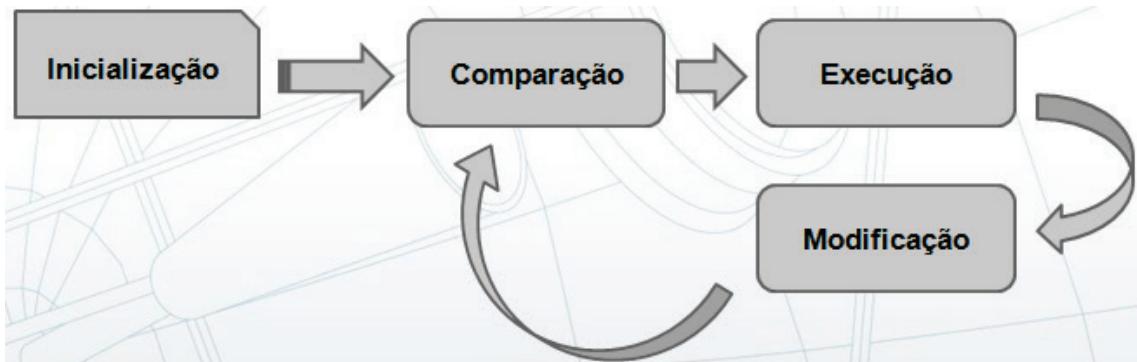
Anotações

Índice

1. Controle de repetições: FOR
 2. Controle de repetições: WHILE e DO-WHILE
 3. Arrays estáticos
 4. C-Strings
 5. Variáveis globais
 6. Exemplo

Anotações

1.1 Controle de repetições: FOR



```
for ( inicia var; compara var; modifica var) {  
    operações  
    ...  
    mais operações  
}
```

Anotações

- Repetição do tipo PARA-FAÇA
- Parâmetros:
 - Inicialização (Declaração)
 - Condição (Comparação)
 - Modificação (Incremento / Decremento)
- Break / Continue

O primeiro tipo de controle de repetição que veremos será utilizando o comando FOR, o mesmo comando PARA-FAÇA visto anteriormente na aula de lógica.

O objetivo do comando FOR é determinar que um trecho de código, ou bloco, será executado repetidas vezes. Para isso o comando FOR precisará de 3 parâmetros, vejamos:

1. Primeiro parâmetro: Inicialização

No parâmetro de inicialização do comando FOR, podemos declarar ou declarar e inicializar variáveis que serão utilizadas no comando FOR. O objetivo geralmente é criar um ou mais contadores que possuirão valores iniciais, tais valores serão alterados até alcançarem um valor limite que indicará o fim da repetição do comando FOR.

Uma observação importante: Variáveis declaradas dentro do comando FOR só existirão dentro do mesmo, ou seja, pertencem ao escopo do comando FOR e estarão indisponíveis no restante da função.

Anotações

2. Segundo parâmetro: Condição

O segundo parâmetro deve ser uma operação de comparação ou um conjunto de operações usando operadores booleanos que retorno verdadeiro ou falso. A execução da repetição acontecerá enquanto a avaliação da comparação for verdadeira.

3. Terceiro parâmetro: Modificação

O terceiro parâmetro consiste em uma operação que altere o valor das variáveis declaradas no primeiro parâmetro e comparadas no segundo. Tipicamente são usados os operados de pré-incremento, pós-incremento ou então incremento com atribuição ou decremeno com atribuição. Veja como funciona na prática.

Detalhe importante. Todos os parâmetros do comando FOR são opcionais e é muito comum encontrar casos onde algum dos três parâmetros não foi utilizado. É bom saber que mesmo nestes casos, é necessário utilizar o ponto-e-vírgula para separar os parâmetros, mesmo que eles estejam vazios.

4. Comandos auxiliares, break e continue

Dois comandos auxiliares podem ser utilizados dentro do FOR. O comando BREAK pode ser utilizado para encerrar o FOR precocemente, caso seja necessário. O comando CONTINUE, por outro lado encerra apenas a repetição atual FOR, executando o incremento e voltando a fazer a comparação para o próximo passo da repetição.

Anotações

1.2 Controle de repetições: WHILE e DO-WHILE

- Repetição do Tipo “ENQUANTO”
- Um parâmetro: Condição de Execução
- WHILE: Compara e Executa
- DO-WHILE: Executa e depois Compara



Anotações

```
inicia var;  
  
while (compara var) {  
    operações  
    ...  
    mais operações  
  
    modifica var;  
}
```

```
inicia var;  
  
do {  
    operações  
    ...  
    mais operações  
  
    modifica var;  
} while (compara var);
```

Em lógica de programação vimos o comando ENQUANTO, capaz de executar um trecho de código de forma repetida até uma condição de parada ser executada. Em C++ possuímos duas versões do comando Enquanto. O comando WHILE e o comando DO-WHILE. Veja a estrutura básica dos dois:

A diferença entre WHILE e DO-WHILE está no momento em que a condição é verificada. No caso do While a condição é verificada ANTES da execução do bloco de comandos. No DO-WHILE, por sua vez, a comparação é feita ao final de cada execução. Desta maneira no DO-WHILE a execução do bloco é feita sempre pelo menos uma única vez.

Vejamos o exemplo que estudamos no comando FOR feito utilizando primeiro WHILE e depois DO-WHILE.

Anotações

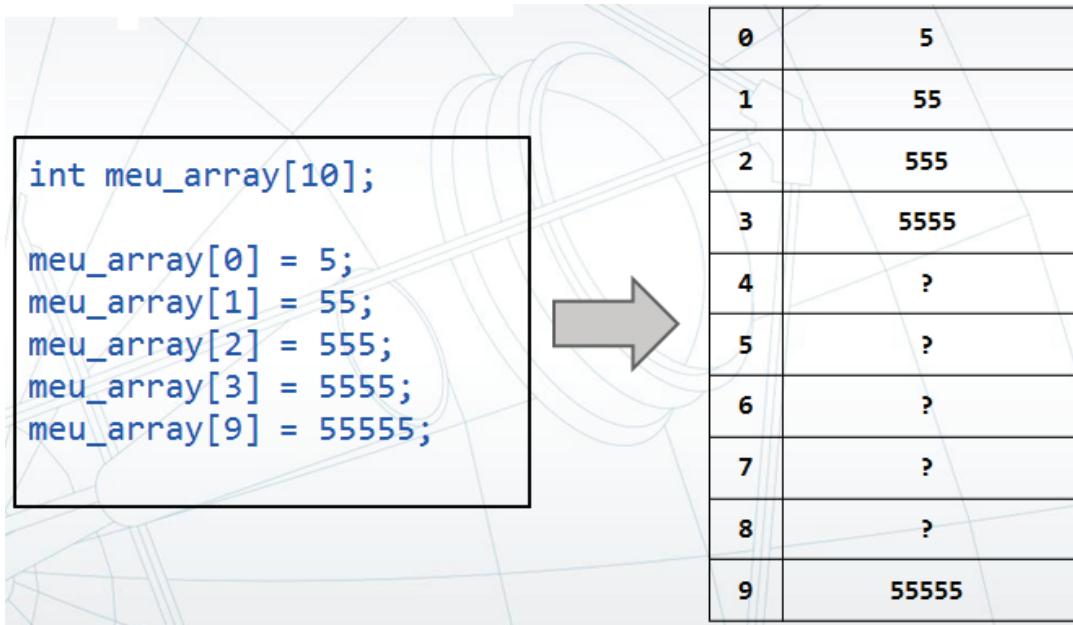
Observação: Tanto o comando break como o comando continue que vimos junto ao FOR possuem a mesma funcionalidade com WHILE e com DO-WHILE.

Em tese, FOR, WHILE e DO-WHILE podem ser utilizados para resolver os mesmos problemas, porém com a prática você perceberá que algumas repetições são mais facilmente representadas em um do que no outro.

1.3 Arrays estáticos

- Sequência de valor do mesmo tipo
- Acessados através da posição
- Posição inicial é zero

Anotações



- Não há verificação no acesso, cuidado com os limites!

Em C++ os arrays estáticos são o tipo mais simples de estrutura sequencial. Todos os elementos do array terão o mesmo tipo de dado e são acessados através de sua posição, que inicia em zero e vai até o número anterior ao tamanho do array.

Anotações

Pontos importantes:

- Após o array ser criado seus valores são indefinidos até que sejam inicializados
- Não há nenhuma validação no acesso ao array em uma determinada posição, logo é preciso ficar atento para não tentar ler ou gravar dados fora do tamanho do array

É muito comum o uso do comando FOR para ler ou gravar dados em um array.

1.4 C-Strings

- Representação básica de texto
- Forma literal: “uma linha de texto”
- Como variável: char minha_string[];
- Caracteres especiais: uso de escape
- Caracteres especiais

Anotações

\0	Final da string	\r	Quebra de Linha Mac
\r\n	Quebra de linha Windows	\t	Tabulação
\n	Quebra de linha Unix	\"	Aspas duplas

- Alerta: Evitar uso quando possível (veremos String STL na próxima aula)
 - Funções adicionais (como concatenação): usar biblioteca cstdlib

A maneira mais básica de representar texto na linguagem C++ é utilizando C-Strings, ou seja, a strings herdadas da linguagem C. A utilização mais casual das C-Strings ocorre naturalmente quando expressamos literais utilizando texto entre aspas no código do nosso programa. Todo texto escrito em aspas duplas é armazenado pelo compilador como uma C-String. Formalmente, cada C-String é um array de caracteres cujo final é determinado pelo caractere de valor numérico zero. Vejamos os exemplos:

Caracteres especiais:

Na declaração de uma string podemos utilizar sequenciais de escape para indicar caracteres especiais, vejamos os mais importantes:

Alerta:

O programador C++ deve conhecer as strings C, porém sua utilização em programas reais deve ser limitada, dando preferência a um tipo mais completo que veremos nas próximas unidades. Tal recomendação se deve ao fato das strings básicas não possuírem gerenciamento automático de memória e limites de acesso aos caracteres.

Anotações

Utilização:

Tanto o acesso quanto a modificação de C-Strings é idêntico aos arrays normais. Strings declaradas literalmente dentro do programa devem ser somente-leitura. E operações mais complexas como concatenação, cópia e strings parciais deve ser feitas preferencialmente utilizando a biblioteca `cstring` (ou `string.h`).

1.5 Variáveis globais

- Disponíveis em todas as funções
- Declaradas no arquivo de cabeçalho
- Inicializadas no arquivo de código fonte
- Criadas antes da função main (e destruídas depois do retorno do main)
- Possuem valor padrão (zero para números)

Variáveis globais permitem o compartilhamento fácil de dados comuns em diversos pontos do programa. As primeiras linguagens de programação não possuíam o conceito de função e logo todas as variáveis eram acessíveis de todo o código do programa, ou seja, eram globais. Alguns detalhes importantes:

- As variáveis globais devem ser apenas declaradas nos arquivos de cabeçalho, para serem acessíveis em diversos arquivos de código fonte.

Anotações

- A variável global declarada no header deve ser inicializada (ou definida) no arquivo de código fonte correspondente e apenas nele. Arquivos de código fonte diferentes não podem redefinir uma variável global com o mesmo nome, para tal deve-se usar o modificador EXTERN na declaração da variável.

Alguns detalhes importantes:

1. As variáveis globais são criadas antes do inicio da função main() e destruídas depois, tal informação será importante para a aula de orientação a objetos
2. Diferentemente das variáveis locais, as globais tem valor padrão zero caso não sejam inicializadas

A utilização das variáveis globais é idêntica as variáveis locais, sendo que a única diferença é sua declaração fora do corpo de uma função específica. É bom lembrar que a prática moderna de desenvolvimento de software advoga a utilização mínima de variáveis globais em favor de outras técnicas.

1.6 Exemplo

- Refazer o Exemplo de Lógica de Programação usando C++
 - Parte 1: Ler e exibir alunos
 - Parte 2: Calcular média e indicar aprovação

Agora vamos escrever um programa em C++ completo. Temos todo conhecimento necessário para escrever o exemplo feito na aula de Lógica da Programação utilizando apenas C++.

Anotações _____

O objeto é ler uma lista de alunos, cada um com 3 notas, calcular o resultado da média e exibir se cada aluno foi aprovado.

Na primeira parte faremos a estrutura do programa e a leitura dos dados e na segunda a exibição dos aprovados.

Anotações

Aula 4 - Estruturas básicas da linguagem C++ parte 3

1. Primeiros passos: Variáveis, Funções e Operações
2. Laços condicionais, estruturas sequenciais e variáveis globais
3. Memória dinâmica, ponteiros e referências
4. Estruturas de dados e acessos a arquivos

1 Memória dinâmica, ponteiros e referências

Nesta unidade estudaremos os seguintes tópicos:

- 1.1 Ponteiros
- 1.2 Referências
- 1.3 Passagem de parâmetro por ponteiro
- 1.4 Passagem de parâmetro por referência
- 1.5 Array alocado dinamicamente (new)
- 1.6 Destruição do array / Gerência de memória

Anotações _____

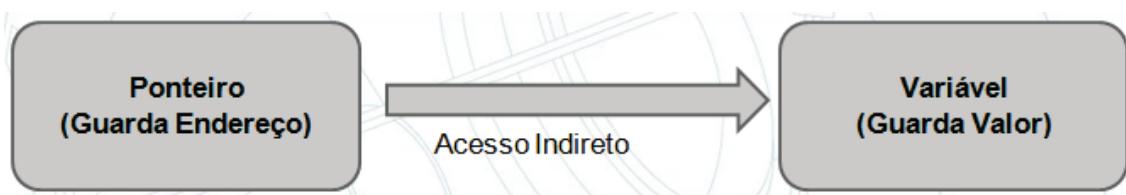
O objetivo desta unidade é aprender o básico sobre gerenciamento e acesso a memória. Tal conhecimento é fundamental na estruturação de programas mais complexos e na criação de programas eficientes.

Índice

1. Ponteiros
2. Referências
3. Passagem de parâmetro por ponteiro
4. Passagem de parâmetro por referência
5. Array alocado dinamicamente (new)
6. Destrução do array / Gerência de memória
7. Exemplo

Anotações

1.1 Ponteiros



```
tipo* nome_do_ponteiro; // forma genérica  
  
int * ponteiro_para_int; // declaração  
int * pont = NULL; // NULL é um ponteiro vazio  
  
int var;  
int * pont = &var; // &var retorna o endereço de var  
  
*pont = 10; // *pont acessa o valor da variável apontada  
cout << *pont;
```

Anotações _____

Ponteiros são um tipo diferenciado de variável em C++ que tem sua origem na linguagem C e utilização idêntica. O objetivo da utilização de ponteiros é o acesso indireto a um determinado dado, ou seja, o ponteiro não guarda o valor da variável, mas sim o local onde ela se encontra. Dessa forma um ponteiro pode apontar para locais diferentes ao longo de sua existência.

De uma forma simples, declaramos e usamos o ponteiro da seguinte forma:

1. Junto ao ponteiro utilizamos 2 operadores unários
2. O operador & é chamado de operador de referência quando usado antes do nome de uma variável. A função do operador de referência é retornar um ponteiro para a variável referenciada.
3. O operador * é chamado de operador de-referência, quando usado na frente de uma variável do tipo ponteiro ele acessa o valor, ou seja, o dado real, da variável que está sendo referenciada

Detalhe importante:

4. A utilização de ponteiros deve ser feita sempre com cautela, quando não houver outro recurso mais adequado

Anotações

1.2 Referencias

Parecida com ponteiros, exceto:

1. Surgiu no C++ e não no C
2. Inicialização obrigatória
3. Não pode referenciar outra variável depois de inicializada
4. Utilização mais simples (sem os operadores adicionais)

Anotações

```
tipo& nome_da_referencia(var); // forma genérica  
  
int & ref_para_int(variavel); // declaração  
int & ref_para_int = variavel; // alternativa  
  
int var;  
int & ref(var); // parece função, mas não é  
  
ref = 10; // ref acessa o valor da variável referenciada  
cout << ref; // não precisa de operador *
```

As referências são um tipo de variável parecida com os ponteiros que vimos no tópico anterior. A primeira diferença é que elas não existiam na linguagem C, foram inventadas especialmente no C++.

5. A segunda diferença é que uma variável do tipo referência precisa ser sempre inicializada na declaração
6. A terceira diferença é que após a inicialização de uma referência, ela irá apontar sempre para a mesma variável

Anotações

7. A quarta diferença é que a utilização, ou seja, o acesso e a modificação dos valores da variável apontada pela referência não requer nenhum operador especial. Assim o uso da referência se torna extremamente simples

Considerações:

8. Sempre que possível, deve-se preferir a utilização de referências no lugar de ponteiros
9. A tentativa de utilizar de forma incorreta uma referência gera erros já na hora da compilação. Com ponteiros é comum o problema só ser descoberto na hora da execução do programa.
10. O verdadeiro poder dos ponteiros e das referências está na passagem de parâmetros, como veremos no próximo tópico.

É importante neste momento você tentar repetir os pequenos exemplos feitos até agora. Modifique nomes e valores para entender bem o que está ocorrendo. Insira erros propositais para verificar quais mensagens o compilador gera em cada problema.

Anotações

1.3 Passagem de parâmetro por ponteiro

Utilidade:

1. Retorno de múltiplos valores
2. Permitir a função modificar uma variável fora de seu escopo
3. Passar dados complexos sem fazer cópias
4. A passagem de parâmetros utilizando ponteiro é um recurso poderoso da linguagem C++ que permite, entre outras coisas:
 5. Simular uma função com retorno de vários valores
 6. Deixar que uma função modifique o valor de uma variável fora de seu escopo
 7. Passar dados complexos (que veremos em orientação a objetos) sem fazer cópias adicionais

Considerações:

1. É uma boa prática verificar os parâmetros fornecidos para evitar manipular um NULL pointer, o que irá provavelmente travar o programa
2. Voltaremos para ver mais detalhes da passagem por ponteiro nas próximas unidades, fazendo uso de modificadores

Anotações

1.4 Passagem de Parâmetro por referência

1. Mesma utilidade da passagem por ponteiro
2. Mais segura / Não tem como passar ponteiro NULL
3. Mais fácil / Não precisa mexer na chamada de função
4. Erros aparecem durante a compilação

As mesmas diferenças vistas entre ponteiros e referências também estão presentes na passagem de parâmetro por referência. A utilização também serve para os mesmos objetivos, porém podemos considerar a passagem por referência como sendo mais segura e menos sujeita a erros.

Dê preferência a passagem por referência sobre a passagem por ponteiro. Sua utilização é mais fácil e menos sujeita a erros.

Anotações

1.5 *Array alocado dinamicamente (new)*

1. Utilização do operador NEW
2. Tamanho definido durante a execução
3. Criado fora do escopo da função
4. Não é apagado automaticamente quando a função retorna

A alocação de memória dinâmica é um dos aspectos mais importantes do desenvolvimento em C++. Alocar memória de maneira dinâmica significa pedir para o sistema operacional que reserve uma determinada região de memória RAM para o armazenamento de uma variável. Tal procedimento será feito através do operador NEW. Neste primeiro contato veremos apenas como criar arrays de forma dinâmica, voltaremos ao assunto novamente na aula de orientação a objetos.

Qual a necessidade de criar variáveis, no caso arrays, de forma dinâmica?

1. Podemos especificar o tamanho do array durante a execução
2. O array criado não será apagado quando perder o escopo em que foi criado
3. O tamanho do array criado com o operador new pode ser muitas vezes maior do que o array local a uma função
4. Mostrar new int[]
5. Mostrar passagem e retorno

Anotações

6. Mostrar valor padrão no new

Como exercício. Faça um programa que crie arrays dinamicamente em um loop infinito e observe no Gerenciador de Tarefas a área total de memória usada pelo programa crescer até ele possivelmente travar. Tente descobrir também qual o maior array que você consegue alocar.

No próximo tópico veremos como gerenciar a vida da memória alocada dinamicamente, ou seja, como desalocar a memória reservada para uma variável

1.6 Destrução do array /Gerência de memória

1. Utilização do operador DELETE
2. Toda memória reservada com NEW deve em algum momento ser liberada como DELETE
3. Cuidados:
4. Não usar delete de forma duplicada
5. Determinar o proprietário: Que parte do código deve alocar e desalocar a variável
6. Não usar variável depois de apagada (Não gera erro de compilação)

Anotações

Na sua forma básica, C++ é considerada uma linguagem não-gerenciada. Em parte, isso significa que toda memória alocada dinamicamente pelo programador precisa ser gerenciada por ele próprio. Quem já é desenvolvedor em outras linguagens de programação costuma estranhar, pois C++ não possui coleta automática de lixo, diferentemente de praticamente todas as linguagens recentes.

Para gerenciar a memória, precisamos utilizar os operadores NEW, e DELETE.

O uso do operador delete é bem simples. No uso prático devemos levar em conta os seguintes pontos:

1. Podemos usar o delete em uma variável em qualquer momento após a sua criação
2. Não devemos deletar 2 vezes a mesma variável
3. Devemos determinar exatamente que parte do nosso programa é a dona da informação. Geralmente o proprietário é responsável por criar e por apagar. Tal conceito também será muito importante na aula de orientação a objetos.
4. Nada nos impede de usar a variável depois que ela for deletada, portanto tenha muito cuidado

Uma sugestão de pesquisa: Faça uma busca por fragmentação interna de memória, para saber um pouco mais sobre um dos problemas que podem ocorrer com o excesso de alocação e desalocação durante a execução de um programa.

Repete o exercício do tópico anterior sobre a alocação em loop, adicione agora também a desalocação com o delete e observe o comportamento da memória total do programa usando o Gerenciador de Tarefas.

Anotações

1.7 Exemplo

Refazer o exemplo da leitura de notas e cálculo de médias usando passagem de referências, ponteiros e alocação dinâmica.

Como exercício, modificar o exemplo visto na unidade anterior. Porém agora fazer a alocação dos arrays de forma dinâmica e utilizar passagem por ponteiro e por referência.

Anotações

Aula 5 - Estruturas Básicas Linguagem C++ Parte 4

1. Primeiros passos: Variáveis, Funções e Operações
2. Laços condicionais, estruturas sequenciais e variáveis globais
3. Memória dinâmica, ponteiros e referências
4. Estruturas de dados e acessos a arquivos

1 Estruturas de dados e acessos a arquivos

1. Vetores
2. Listas
3. Iterador
4. Iterador reverso
5. Mapas
6. Iterador em mapas
7. Strings STL - Criação
8. Strings: Substring e concatenação

Anotações

9. Abertura de arquivos e leitura
10. Escrita em arquivos
11. Exemplo

Nesta unidade veremos as principais estruturas de dados utilizando a biblioteca STL, na sequencia, também utilizando a STL, veremos uma forma mais poderosa e segura de utilizar strings, teremos também uma pequena amostra do acesso de leitura e escrita em arquivos e no final um exemplo complexo englobando todos esses conceitos.

Prepare um novo projeto em branco no QtCreator e não esqueça de testar todos os pequenos exemplos mostrados em cada tópico e por último, repetir o exemplo final da unidade. Mão a obra!

1.1 *Vetores*

1. “Array melhorado”
2. Utilização semelhante
3. Redimensionamento automático e manual
4. Gerenciamento e controle dos limites
5. Possibilidade de passagem por cópia

Anotações

- Indicado:
 1. Validação no acesso
 2. Acesso sequencial
 3. Comprimento constante
- Contra-indicado:
 1. Remoção frequente
 2. Inserção no início ou no meio
- Vantagem:
 1. Pode ser usado como se fosse um tipo primitivo

Anotações

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

vector<int> vet(10);

vet[0] = 15;
vet[1] = 20;

vet.push_back(100);
```

A primeira estrutura de dados vista será o vetor. De uma maneira simples, o vetor pode ser visto como um array cujo tamanho se ajusta automaticamente às nossas necessidades. Sua utilização básica também é idêntica ao array, utilizando o mesmo operador para acesso. Diferentemente do array padrão do C++, os vetores são fornecidos pela biblioteca STL, utilizando o include <vector>.

Anotações

A utilização do vetor é indicada nos seguintes casos:

- Para substituir arrays por uma implementação mais segura. Os vetores possuem validação no acesso, não permitindo ler dados que ainda não foram inseridos.
- Quando precisamos de uma estrutura com acesso sequencial rápido e pouca (ou quase nenhuma) necessidade de remover dados ou inserir dados no meio da estrutura.

A utilização não é recomendada para:

- Estruturas que sofrem muita modificação durante a execução, principalmente remoção de dados do início da estrutura

E a principal vantagem:

- Os vetores podem ser tratados como variáveis primitivas, podendo ser passados como parâmetro através de cópia e não de referência ou ponteiro quando necessário

1.2 *Listas*

1. Encadeadas
2. Suportam mais elementos que vetor

Anotações

3. Vantagem
 - 3.1 Remoção aleatória
 - 3.2 . Inserção aleatória
4. Desvantagem
 - 4.1 Acesso aleatório
 - 4.2 Uso diferente dos vetores e array

Anotações

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

list<int> lst;

lst.push_back(100);
lst.push_front(200);

int a = lst.back();
```

Listas são estruturas encadeadas, diferentemente dos arrays e vetores que são estruturas sequenciais. Isso significa que internamente, os dados de uma lista não estão em posições consecutivas na memória do computador. Tal diferença possui as seguintes vantagens:

Anotações

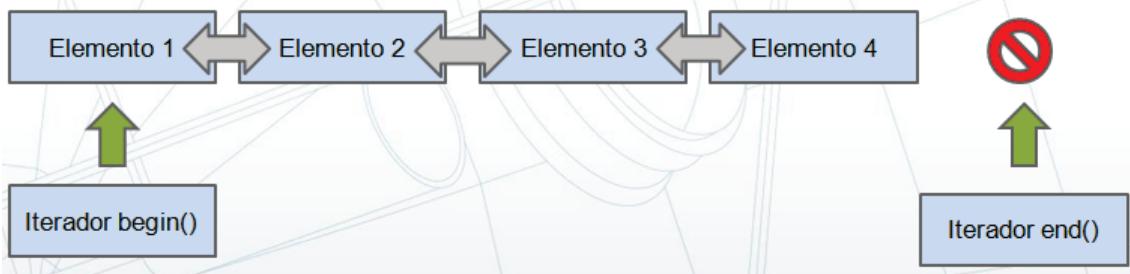
1. As listas comportam mais dados do que os vetores, já que nem sempre o sistema operacional consegue alocar áreas contínuas grandes de memória para o programa
2. A alteração aleatória de uma lista é mais eficiente, principalmente a remoção / inserção de dados no início e no final da lista.
3. Podemos destacar também algumas desvantagens:
4. O uso das listas é mais complicado, não sendo possível utilizar o operador [] para acessar um elemento na posição especificada.

Veremos agora como declarar e fazer algumas operações na lista. O acesso mais completo será visto no próximo tópico, utilizando iteradores.

Anotações

1.3 Iterador

Acesso / Navegação dos elementos de uma estrutura



1. Operações
 - 1.1 Deslocamento: iterador++, ++iterador, iterador += numero, iterador--
 - 1.2 Acesso ao elemento: *iterador
 - 1.3 Comparação: Apenas (==) e (!=)
 - 1.4 Criação: estrutura.begin()
2. Detalhe:
 - 2.1 Iterador end() é constante, não pode ser deslocado, apenas comparado!

Anotações

Iteradores são fundamentais no uso de estruturas de dados em C++. Todas as estruturas da STL implementam o conceito de iterador. Qual seria esse conceito então?

Chamamos de iterador um tipo de variável particular a cada estrutura de dado que permite navegar, ou percorrer a estrutura, avançando ou recuando dentro dela.

O iterador geralmente será utilizado em um laço condicional, tipicamente o operador FOR.

Operações importantes dos iteradores:

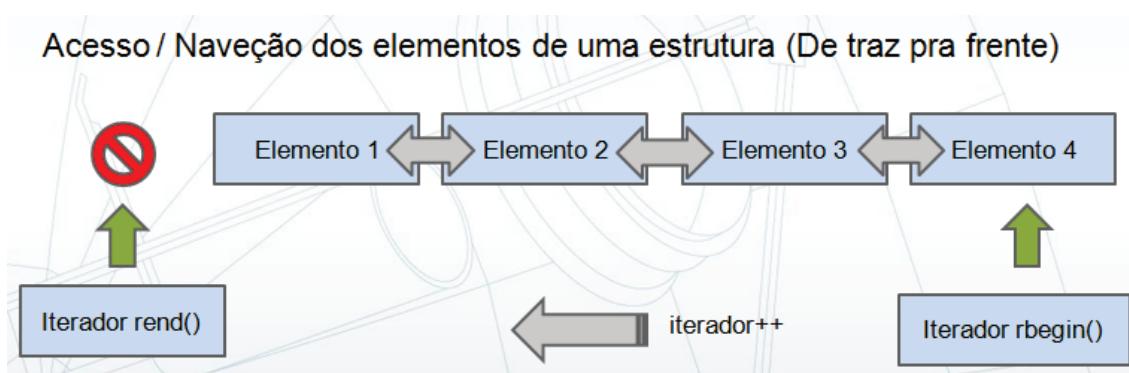
1. Incremento, decremento ou soma e subtração: Para deslocar o iterador pela estrutura de dados
2. De-referencia, utilizando o operador asterisco: permite ler ou modificar o dado da estrutura onde o iterador se encontra
3. Comparação com outro iterador, para saber se ambos estão na mesma posição ou não: Apenas o operador IGUAL e DIFERENTE pode ser usado. Um erro comum é tentar comparar iteradores com os operadores maior que e menor que.
4. Operação estrutura.begin() que retorna o iterador para o inicio da estrutura.

Detalhe importante:

- A operação estrutura.end() retorna um iterador constante para o fim da estrutura. Esse iterador não representa o último elemento da estrutura de dados, mas sim o final da estrutura onde não resta mais nenhum elemento.

Anotações

1.4 Iterador reverso



Iteradores reversos tem como o objetivo percorrer a estrutura de trás para frente. Sua utilização é basicamente identica ao iterador convencional. O que muda são as operações para acessar o inicio e o final da estrutura. Veja:

1. A operação estrutura.rbegin() retorna um iterador reverso que inicia no último elemento da estrutura
 2. A operação `++` no iterador faz ele andar em sentido ao início da estrutura
 3. A operação estrutura.rend() retorna um iterador para o inicio da estrutura, após o primeiro elemento. Ou seja, quando a estrutura está vazia `.rbegin() == .rend()`

Anotações

1.5 Mapas

1. Armazena pares
2. Utilização como uma tabela CHAVE \Rightarrow VALOR
3. Vantagens
4. Diversas chaves
5. Inserção / Remoção rápida
6. Uso fácil
7. Crescimento automático

Anotações

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

map<float, int> mp;

mp[5.1] = 10;
mp[10.2] = 20;

mp.erase(5.1);
```

Mapas são estruturas que representam um par de dados, diferente das demais estruturas vistas até agora. O primeiro elemento do par é chamado de chave, e precisa ser único da na estrutura. O segundo elemento é chamado de valor e tem seu acesso atrelado ao primeiro elemento, que é a chave.

Internamente os mapas são estruturas em formato de árvore, também encadeadas. Onde o acesso a um determinado valor através da sua chave pode ser feito usando o operador [], tal como num array. As vantagens do mapa são:

Anotações

1. Possibilidade de usar chaves diversas, como strings, ponto flutuante, caracteres.
2. Acesso rápido a pontos aleatórios da estrutura, a organização em árvore permite um acesso mais inteligente aos dados, não sendo necessário percorrer a estrutura inteira
3. Utilização simples, tal como nos arrays e vetores
4. Crescimento automático

1.6 Iterador em mapas

1. Retorna um par
 - Primeiro elemento =>Chave
 - Segundo elemento =>Valor
2. Percorre o mapa sempre de maneira ordenada
3. Uso:
 - 3.1 iterador->first (chave)
 - 3.2 iterador->second (valor)

Os mapas diferem mais das outras estruturas pois seu iterador retorna uma dupla, ou um par, ao invés de retornar apenas o valor. Logo precisamos especificar qual dos dois elementos queremos.

Anotações

Temos então:

- iterador->first = acessa a chave do elemento atual no iterador
- iterador->second = acessa o valor do elemento atual no iterador

1.7 *Strings STL –Criação*

1. Aperfeiçoamento das C-strings
2. Podem ser usadas como tipo primitivo
3. Gerenciamento automático
4. Redimensionamento
5. Checkagem de limites
6. Comparação usando operadores (==, !=, > e <)
7. Criação / Conversão para C-string

As strings em STL são uma evolução das C-strings, oferecendo as seguintes vantagens:

1. Tratamento como tipo primitivo: Não precisamos nos preocupar com a alocação e desalocação
2. Cópia, redimensionamento e validação no acesso. Tudo de maneira transparente e automática

Anotações

3. Comparação simplificada utilizando os operadores IGUAL, DIFERENTE, MAIOR E MENOR
4. Possibilidade de iniciar a partir de uma string C e operação para retornar um string C (oferecendo compatibilidade com bibliotecas que não aceitem strings STL).

1.8 *Strings: Substring e concatenação*

1. Concatenação com operador (+) gera nova string
2. Concatenação com operação append() modifica string original
3. Operação de substring (substr) gera nova string
4. Operação de substituição (replace) pode trocar trecho da string original

Diferentemente das strings em C, as strings STL possuem operações de substring e de concatenação que podem não modificar as strings originalmente envolvidas, ou seja retornam strings novas.

Anotações

Ambas as operações são feitas da seguinte forma:

1. A concatenação pode ser feita utilizando o operador de adição (+)
2. A concatenação modificando a string original deve ser feita utilizando a operação append()
3. A extração de uma substring é feita com a operação substr(), enquanto que a modificação de uma substring pode ser feita utilizando a operação replace() ou acessando cada caractere individualmente com o operador []

1.9 Abertura de arquivos e leitura

1. Abertura utilizando nome e modo de utilização
2. Escrita e leitura no mesmo estilo CIN e COUT
3. Operação stream.eof() indica final do arquivo

Anotações

Leitura	fstream::in	Append (Escrita no final)	fstream::app
Escrita	fstream::out	Truncate (Zera tamanho do arquivo)	fstream::trunc
Modo binário	fstream::binary		

Utilizamos também a STL para fazer leitura e escrita em arquivos. Tais operações são feitas através do include FSTREAM. Uma FSTREAM tem o comportamento parecido da IOSTREAM, quando usamos as streams CIN e COUT para leitura e escrita no console. A grande diferença está na inicialização e finalização, ou seja, as FSTREAMS para ler ou escrever em um arquivo precisam ser abertas e fechadas. Sendo que no processo de abertura especificamos qual o arquivo que estamos interagindo e qual o modo que essa iteração será feita

1. Abrimos as streams com a operação open seguida do nome como uma string ou C-string
 2. Especificamos o modo, de acordo com esta tabela, utilizando o operador OU-binário
 3. Podemos testar o final do arquivo usando fstream.eof()

Após a abertura, podemos ler a partir do arquivo tal como fazemos a leitura a partir do console.

Para finalizar: Não existe a necessidade de fechar o arquivo de maneira explícita sempre. Isso se deve ao fato de quando a variável `fstream` perder o escopo, sua destruição implica automaticamente no fechamento do arquivo por ela aberto.

Anotações

1.10 *scrita em arquivos*

1. Uso identico ao COUT
2. Modo binário => Não converte quebras de linha
3. Append => Começa escrita a partir do fim do arquivo
4. Teste de integridade stream.fail() e stream.bad()

Leitura	<code>fstream::in</code>	Append (Escrita no final)	<code>fstream::app</code>
Escrita	<code>fstream::out</code>	Truncate (Zera tamanho do arquivo)	<code>fstream::trunc</code>
Modo binário	<code>fstream::binary</code>		

A escrita em arquivos também é análoga a escrita na tela. O mais importante neste caso são observar os parâmetros de abertura do arquivo, sua definição correta é muito importante:

1. A opção binary indica que o arquivo será gravado de forma binário, ou seja, a STL não irá fazer conversão automática da quebra de linha \n para \n\r no Windows.
2. A opção append indica que a escrita deverá iniciar no final do arquivo, não no começo.

Anotações _____

3. E por ultimo, a opção truncate indica que o arquivo deve ter seu conteúdo apagado e a escrita iniciará do começo

O uso da escrita ficará identico ao do COUT.

Problemas na abertura do arquivo ou na escrita ocasionados por falhas, como disco cheio ou remoção de um dispositivo, podem ser checadas usando fstream.fail() e fstream.bad().

1.11 Exemplo

1. Agenda telefônica
 - 1.2 Cadastro e exibição
 - Exclusão
 - Salvar em arquivo
 - Pesquisa por nome

Anotações _____

Exemplo dividido em 4 partes.

A primeira consistirá na entrada de dados para uma lista telefônica com numero de telefone e nome.

Na segunda parte implantaremos um comando que permita a remoção de itens da agenda, a partir do numero de telefone.

A terceira parte será um comando para gravar o conteúdo da agenda em um arquivo especificado pelo usuário.

Por ultimo, criaremos um comando de pesquisa usando o nome.

Anotações

Aula 6 - Orientação a Objetos em C++

1. Organização do Software Orientado a Objetos
2. Aperfeiçoamento do Software Orientado a Objeto

1 Organização do software Orientado a Objetos

A orientação a objetos é um paradigma de desenvolvimento de software nascido na década de 80 como evolução da programação estruturada. Atualmente a maioria das linguagens de programação foi concebida com a orientação a objetos em mente ou possui algum recurso que permite simulá-la.

No caso da linguagem C++, temos que ela foi desenvolvida a partir do C justamente para incorporar os recursos do novo paradigma. É importante frisar que utilizar os recursos de orientação a objetos não significa necessariamente programar orientado a objetos. Ou seja, utilizar o paradigma significa adotar uma metodologia de criação de software e não apenas seus recursos isolados.

Anotações

Com o objetivo de aprender o paradigma orientado a objetos usando a linguagem C++, nossa aula será dividida em 2 grandes partes:

- Organização do software Orientado a Objetos: Que tem como objetivo mostrar os recursos iniciais da orientação a objetos
- Aperfeiçoamentos do software Orientado a Objetos: Unidade em que aprenderemos algumas das funcionalidades mais complexas

Não se esqueça de acompanhar e executar os exemplos, sempre adicionando pequenas modificações para evoluir o aprendizado com seus próprios acertos e erros.

1. Classes, Objetos, Atributos e Métodos
2. Encapsulamento - Public
3. Encapsulamento - Private
4. Instanciação
5. Construtores
6. Destrutores
7. Herança
8. Encapsulamento - Protected
9. Exemplo

Anotações

1.1 *Class, Objetos, Atributos e Métodos*

1. Programação Orientada a Objetos (POO)
2. Código dividido em Classes
3. Classe, define:
 - 3.1 Atributos: Variáveis internas
 - 3.2 Métodos: Funções no escopo de uma classe
4. Objeto \Rightarrow Pertence a uma classe, atributos possuem valores próprios

No desenvolvimento estruturado, nos preocupamos em dividir o programa em funções e armazenar os dados em variáveis globais e locais. O principal problema surge quando o número de funções e de variáveis globais se torna muito grande. Começa a ficar difícil definir quais funções fazem parte de determinado componente do programa, também é difícil determinar quais funções modificam determinada variável global. Tais problemas são de difícil solução e costumam envolver convenções não-formais, ou seja, apenas regras de boas práticas dentro de uma equipe de desenvolvimento. A orientação a objetos surge para solucionar este problema através de uma nova maneira de organizar o desenvolvimento e a execução de um software.

O elemento principal da escrita de um software orientado a objetos deixa de ser a função e passa a ser a classe. Uma classe é definida pelo programador como um conjunto de atributos e métodos. Durante a execução do programa serão criados os objetos, sendo que cada objeto é uma instância de uma determinada classe.

Anotações

Temos então:

1. classe: código definido pelo programador que determina os atributos e métodos, sendo que cada programa pode ter inúmeras classes
2. atributo: variável interna da classe, possuindo nome e tipo e tendo como escopo a própria classe
3. método: função cujo escopo é a classe e que possui como parâmetro implícito o objeto em que o método é executado (variável this)
4. objeto: realização da classe, que possui valores próprios para cada atributo. Cada classe poderá ter inúmeros objetos criados a partir dela e com valores internos próprios para cada atributo

Além da organização, também são objetivos da orientação a objetos o encapsulamento e o reuso, que veremos com detalhes nos tópicos seguintes.

Anotações

1.2 *Encapsulamento —public*

1. Encapsulamento: Restrição do acesso aos membros de uma classe (atributos e métodos)
2. Tipos:
 - 2.1 Public
 - 2.2 Private
 - 2.3 Protected
3. Public: Acesso livre para código fora da mesma classe

Quando escrevemos classes, torna-se interessante especificar quais membros (atributos e métodos) da classe serão acessíveis por outras classes. O nome dessa separação, entre o que é visível e o que não é, para outras classes do mesmo programa, chama-se encapsulamento.

O encapsulamento em C++ é determinado por palavras que definem a visibilidade, sendo elas:

1. public
2. private
3. protected

Veremos a visibilidade public, na sequencia a private e após o tópico sobre herança, a visibilidade protected.

Anotações

1.3 Encapsulamento —Private

1. Encapsulamento padrão
2. Pode ser especificado também com uma seção “private”
3. Atributos e métodos só podem ser utilizados pelo código da própria classe

Dentro da definição de nossa classe, a visibilidade padrão dos membros (atributos e métodos) é a visibilidade private, mesmo que esta não esteja especificada. Logo, quando começamos a declarar os primeiros atributos, caso não seja definido o início da seção public, estes atributos serão definidos como private.

No entanto, caso haja o interesse de, após a seção public, especificar mais atributos ou métodos como private, é possível fazê-lo usando a palavra “private:”. Todos os métodos e atributos definidos como private terão seu uso restrito internamente e só poderão ser usados pelos métodos da própria classe.

1.4 Instanciação

1. Criação de um objeto a partir de uma classe
2. Feita implicitamente na declaração da variável

Anotações _____

3. Feita pelo operador NEW quando o objeto é alocado dinamicamente

Durante a execução do programa, os objetos precisam ser criados e destruídos, tal como aprendemos, por exemplo, com arrays e arrays dinâmicos.

Logo, podemos instanciar (criar) os objetos principalmente de duas maneiras:

1. No escopo local, especificando o tipo da variável como o nome da classe e o nome da variável que representará o objeto
 2. Na memória do programa, utilizando o operador NEW para construir e o DELETE para apagar. Como o operador NEW retorna um ponteiro, todas as regras que aprendemos relacionadas com ponteiros também são válidas para ponteiros para objetos.

1.5 Construtores

1. Método especial (tem o mesmo nome da classe)
 2. Inicialização dos atributos do objeto
 3. Parametrizado: Pode receber valores
 4. Importante:
 - 4.1 Construtor vazio é padrão
 - 4.2 Encapsulamento (geralmente deve-se usar public)

Anotações

É possível interferir na criação dos objetos através da definição de métodos construtores. Um construtor é um método especial cujo nome é o próprio nome da classe.

Os construtores podem ser parametrizados, para permitir a criação de objetos fornecendo parâmetros de inicialização para o mesmo. Os construtores parametrizados possuem um recurso especial chamado de lista de inicialização, que deve ser usado para inicializar os atributos.

Detalhes importantes:

1. Caso não seja especificado, toda classe tem um construtor padrão vazio
2. O encapsulamento também se aplica aos construtores, que quase sempre serão do tipo public

1.6 Destruidores

1. Método especial
2. Executado antes do encerramento de um objeto
 - 2.1 Quando a variável com o objeto perde o escopo
 - 2.2 Quando DELETE é usado no ponteiro para o objeto
3. Não pode ser parametrizado

Anotações _____

4. Destruitor padrão vazio
5. Quando é preciso definir um destrutor?
 - 5.1 Para manipular ou destruir outros objetos relacionados

Destruidores são o complemento dos construtores. Também são métodos definidos pelo programador cujo código é executado imediatamente antes da destruição de um objeto, seja pelo operador **DELETE**, seja pelo objeto perder seu escopo.

Temos diferenças:

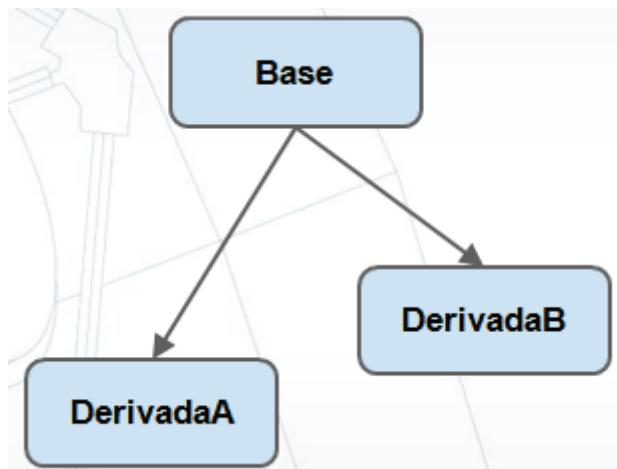
- O destrutor tem o método com o nome da classe, prefixado com o símbolo til
- Não é possível parametrizar o destrutor

Por fim, toda classe possui um destrutor padrão com código vazio. Devemos criar destruidores sempre que a lógica da nossa classe exigir que a destruição de um objeto deva acarretar na destruição ou modificação de outros objetos associados.

Anotações

1.7 Herança

- Reuso de código
- Classe Derivada herda os atributos e métodos da Classe Base
- Importante: Os membros private não são acessíveis às derivadas



Ao lado do encapsulamento, a herança é vista como uma das principais inovações da orientação a objetos. Seu objetivo é o reuso de software, ou seja, evitar a necessidade de reescrever código em partes diferentes do programa.

Anotações

Sua implementação é feita da seguinte maneira:

- Ao declarar uma nova classe, chamada de classe derivada, podemos definir uma classe base. A classe derivada irá possuir tudo que foi declarado na classe base.

A herança, desta forma, permite organizar o programa como uma árvore de classes, onde o conteúdo em comum de classes distintas é unido em uma mesma classe base, facilitando a compreensão e expansão do código fonte do programa.

Ponto importante: Os membros definidos como private na classe base, não poderão ser acessados diretamente pela classe derivada. Veremos como lidar com isso agora no próximo tópico.

1.8 Encapsulamento —Protected

- Encapsulamento protected
 - Funciona como public para as classes derivadas
 - Funciona como private para as demais classes

Já vimos dois tipos de encapsulamento, a visibilidade public e a private. Agora que conhecemos o recurso de herança, precisamos aprender também o encapsulamento protected.

Anotações

O objetivo do `protected` é bem simples, todos os atributos e métodos `protected` são privados para acesso externo, exceto para as classes derivadas. Ou seja, os `protected`s só podem ser usados pela própria classe e pelas classes derivadas dela.

1.9 Exemplo

- Usaremos os recursos de orientação a objetos para fazer uma agenda telefônica com as operações de inclusão, pesquisa, exclusão e listagem.

Vamos escrever um exemplo completo agora, utilizando a metodologia orientada a objetos. Nossa exemplo será implementar uma agenda telefônica, tal como visto na programação estruturada, porém dessa vez usando o máximo possível de recursos orientados a objetos vistos até agora. Não se esqueça de programar junto a partir de um novo projeto!

Anotações

Aula 7 - Orientação a Objetos em C++, Aperfeiçoamentos

1. Polimorfismo: Sobrecarga
2. Polimorfismo: Sobrescrita
3. Polimorfismo: Métodos virtuais
4. Sobrecarga de operador
5. Classe Abstrata
6. Interfaces
7. Composição
8. Agregação
9. Modificador const
10. Modificador static
11. Exemplo

Anotações

1 Aperfeiçoamentos do software orientado a objetos

Nos tópicos seguintes, continuaremos nos aprofundando nos conceitos de orientação a objetos. Os conceitos agora serão um pouco mais avançados, por isso fique atento e sempre que necessário reveja os conteúdos para não acumular dúvidas.

Os próximos tópicos serão:

Funcionalidades adicionais relacionadas aos métodos através do:

- Polimorfismo: Sobrecarga
- Polimorfismo: Sobrescrita
- Polimorfismo: Métodos virtuais

A ampliação da sintaxe da linguagem através da:

Sobrecarga de operador

A implantação dos conceitos de engenharia de software orientado a objetos que envolvem:

- Classe Abstrata
- Interfaces
- Composição
- Agregação

Anotações

O significado especial dos modificadores CONST e STATIC para atributos e métodos de uma classe:

- Modificador const
- Modificador static
 - E por último, um exemplo sintético trabalhando o conteúdo abordado.
 - Exemplo 7

1.1 Polimorfismo: Sobrecarga

- Conceito: Várias funções e métodos com o mesmo nome
- Diferença: Tipo de retorno e lista de parâmetros
- Execução: Determinada durante a compilação
- Pontos importantes:
 - Não gera impacto no desempenho
 - Ambiguidade gera erro de compilação

Anotações

Primeiramente é importante conhecer o conceito de polimorfismo. Entendemos como polimorfismo sempre que algo pode assumir várias formas. No contexto da programação, isso implica em funções ou métodos com o mesmo nome, mas com implementações diferentes. Veremos como tratar cada um dos casos começando pela sobrecarga de métodos e funções.

A sobrecarga consiste em declarar mais de um método de classe ou função com o mesmo nome e com lista de parâmetros diferentes. Durante a fase de compilação do programa, o compilador irá usar a função correta de acordo com os tipos e a quantidade de parâmetros fornecida para o método ou função, veja os exemplos.

O objetivo da sobrecarga é facilitar o desenvolvimento. Sem ela, precisaríamos criar funções com nomes diferentes, mesmo quando o mais natural seria defini-las com o mesmo nome e apenas parâmetros de tipos e quantidades diferentes. Boa parte do código disponível em bibliotecas, como a STL que utilizamos constantemente, faz extenso uso da sobrecarga.

Diferentemente do que muita gente pode falar. O simples uso da sobrecarga não gera nenhum impacto no desempenho do software. O único problema mais grave que pode acontecer é a ocorrência de ambiguidades que impossibilitam a compilação.

1.2 Polimorfismo: Sobrescrita

- Usada junto com a herança
- Classe base ⇒ Implementação original do método

Anotações _____

- Classe derivada ⇒ Pode substituir a implementação, mantendo a mesma assinatura

O polimorfismo por sobreescrita acontece exclusivamente no mecanismo de herança. A idéia básica é a seguinte: Na classe base, declaramos e implementamos um método de determinado nome. A visibilidade do método deverá ser public ou protected. Já na classe derivada, podemos sobreescrita, ou seja, substituir a implementação do método por outra, mantida a mesma assinatura (ou seja, tipos e quantidades de parâmetros).

O recurso da sobreescrita também não gera nenhum impacto de desempenho e diferentemente da sobrecarga não sofre com ambiguidades. As limitações mais importantes são duas:

O código da classe base sempre vai usar a implementação da própria classe.

Se o método sobreescrito for usado em um objeto que sofreu conversão (da classe derivada para a classe base), a implementação da classe base será executada.

Ambas as limitações serão solucionadas com o próximo recurso que veremos e ainda abrirão a possibilidade de implantar os conceitos de classe abstrata e interface, que também serão vistos no futuro.

1.3 Polimorfismo: Métodos virtuais

- Despacho dinâmico ⇒ Método escolhido durante a execução
- Pequeno impacto no tempo de execução

Anotações _____

- Melhoramento do polimorfismo por sobreescrita:

- Código na classe base pode executar método na classe derivada
- Método da classe derivada pode ser usado a partir de um ponteiro ou referência pra classe base

Os métodos virtuais são um recurso da linguagem C++ que implementam o conceito de despacho dinâmico. O despacho dinâmico é a descoberta de qual código executar quando um método for chamado durante a execução do programa. Desta maneira fica claro que os métodos virtuais causam um pequeno impacto no tempo de execução do programa, porém permitem implementar funcionalidades mais sofisticadas.

Com os métodos virtuais poderemos implementar o polimorfismo por reescrita de uma maneira que solucione os dois problemas já mencionados, ou seja: O código da classe base poderá executar um método virtual com implementação feita nas suas classes derivadas e um objeto convertido da classe derivada pra a base poderá ter os métodos sobreescritos executados corretamente.

O polimorfismo com métodos virtuais é fundamental na maioria dos frameworks, inclusive o Qt. Sua correta utilização permite o desenvolvimento de softwares compostos de centenas de classes organizadas de maneira sistemática e com elevada coerência e baixa coesão, ou seja, um software feito por componentes altamente integrados e com baixo grau de dependências.

Anotações

1.4 Sobre carga de operador

- Implementar código para operadores simbólicos:
- + - * == != >= <= ...
- Operandos: Objetos e/ou tipos primitivos
- Exemplo: operadores << e >> junto ao cin e cout
- Utilização controversa!

A sobre carga de operadores é um tipo de polimorfismo por sobre carga, onde no lugar de funções e métodos, implementamos código para os operadores simbólicos do C++, como o operador de soma, subtração, igualdade, maior que, menor que e afins.

Como exemplo, temos a própria utilização os objetos CIN e COUT que vimos até agora, sem discutir o porque da utilização do operador shift-left como operador de saída e o operador shift-right como operador de entrada.

A sobre carga de operadores é um pouco controversa e deve ser utilizada sempre de maneira atenta as convenções estabelecidas nos projetos de software em que trabalhamos. Isso porque alguns desenvolvedores consideram que a utilização da sobre carga de operadores melhora a legibilidade do código, enquanto outros questionam esse argumento e ainda apontam outras desvantagens.

Anotações

1.5 Classe abstrata

- Classe Abstrata ⇒ Pode ser usada apenas na herança
- Implementação:
 - Construtor desabilitado (usando visibilidade private)
 - Método virtual puro (sem implementação)

Durante a modelagem dos nossos programas, vimos a necessidade de agrupar conceitos em classes base e implantar diversas classes derivadas. Essa noção de classe base dá um passo além quando falamos de classes abstratas.

Uma classe abstrata é um tipo de classe que só pode ser usada como base para outras classes, através da herança. Ou seja, não é possível instanciar objetos diretamente de uma classe abstrata. Na linguagem C++ não existe um comando ou modificador para indicar explicitamente que uma classe é uma classe abstrata, sendo que ela irá surgir em duas situações:

- A primeira situação é quando todos os construtores da classe são declarados como private e não há nenhum método que faça o papel de construtor. Desta maneira é impossível criar objetos da classe diretamente.
- A segunda maneira de criar classes abstratas é utilizando declarações de métodos virtuais puros. Um método virtual puro não possui implementação padrão e precisa ser implementado na classe derivada para que esta deixe de ser abstrata.

Anotações

1.6 Interfaces

- Classe Abstrata:
 - Sem nenhum atributo
 - Apenas métodos virtuais puros
- Objetivo ⇒ Contrato de interface (Maneira comum de usar uma classe sem conhecer sua implementação)
- Várias classes implementam uma ou mais interfaces (através de herança)

Interfaces, no contexto da orientação a objetos, são um tipo mais restrito de classe abstrata. Nas interfaces não existem declarações de atributos ou implementações de métodos, ou seja, apenas métodos do tipo virtual puro.

As interfaces são úteis para definir uma forma comum de utilização da classe. Chamamos esse conceito de contrato de interface. Ou seja, parte-se da ideia de que todas as classes que implementam uma determina interface podem ser utilizadas da mesma maneira, para implementar algum conceito.

Anotações

1.7 Composição

- Atributos da classe são objetos de outras classes
- Forma Direta:
 - Declarando os atributos como objetos normais
- Objetos da composição criados e destruídos automaticamente com o objeto compositor
- Forma Indireta:
 - Atributos são ponteiros para objetos
 - Objetos são instanciados no construtor da classe compositora e deletados no destrutor

Ao declarar uma classe, podemos inserir atributos que são objetos de outras classes. Quando fazemos isso, estamos criando uma composição. A composição em C++ pode ser feita de duas maneiras, vejamos:

- Diretamente: Declaramos os atributos com o tipo de classe usada na composição. Durante a criação do objeto da classe em questão, todos os construtores dos atributos que também são objetos vão ser chamados. O mesmo ocorre para os destrutores.
- De forma indireta: Quando usamos ponteiros para objetos no lugar de objetos. Precisamos fazer a instanciação dos ponteiros explicitamente utilizando o operador NEW para cada atributo no construtor de nossa classe e chamar o operador DELETE para os mesmos atributos no destrutor.

Anotações

1.8 Agregação

- Mais fraca que a composição
- Atributos são ponteiros ou referências para objetos
- Não são obrigatoriamente criados ou destruídos na classe agregadora

A agregação é um tipo de vínculo mais fraco que a composição. Ou seja, a classe possui atributos que são referências ou ponteiros para objetos essenciais para sua existência, porém a vida desses objetos, ou seja, sua criação e destruição é controlada externamente a classe.

É importante ressaltar que a composição e agregação são conceitos de software que não precisam refletir a lógica do mundo real, mas sim as decisões de modelagem do software. Ou seja, podemos modelar uma classe Carro, fazendo tanto composição quanto agregação da classe Motor, a decisão independe do Motor poder existir independente do carro na vida real.

Anotações

1.9 Modificador Const

- Atributos const => Não podem ser modificados (constantes)
- Métodos:
 - Retorno const => Objeto retornado não pode ser alterado
 - Parâmetro const => Parâmetro somente-leitura
 - Métodos const => Indicativo que o método não altera o objeto

Modificadores, como visto anteriormente, são palavras especiais que alteram a declaração de uma variável, função ou método. No contexto de uma classe, o modificador const pode ter 4 papéis diferentes.

- Atributos const: São atributos que não podem ser modificados, geralmente são dados numéricos constantes e funcionam como um atalho para que não se precise decorar o valor numérico destas constantes.
- Métodos que retornam const: Nesse tipo de método, o objeto retornado não pode ser modificado. O uso mais comum é retornar uma referência const.
- Métodos que possuem parâmetros const: Tal como as funções, os parâmetros marcados como const serão apenas lidos e não modificados. É uma boa prática marcar todos os parâmetros que são ponteiros ou referência e não são alterados com o modificador const.

Anotações

- Métodos que possuem const após a lista de parâmetros: Estes métodos indicam que não irão modificar o conteúdo do próprio objeto onde eles serão executados. Dessa maneira é possível executar métodos marcados como CONST em objetos marcados como CONST, garantindo assim a sua integridade.

Para se habituar com possíveis erros de compilação, sugiro que você altere o modificador e tente executar o exemplo.

1.10 Modificador Static

- Atributos static:
 - Atributo compartilhado (independente)
 - Não precisa de objeto (pode ser usado com const)
- Método static:
 - Método “de classe” ⇒ Usado sem objeto (não possui this)
 - Prefixado com o nome da classe
 - Pode acessar atributos private de objetos da classe
 - Uso comum para substituir construtores

Anotações

O modificador static no contexto das classes tem apenas dois objetivos, um para atributos e outro para métodos. Veja:

- Nos atributos, o static indica que o atributo é compartilhado entre todos os objetos e pode inclusive ser utilizado sem um objeto associado, ou seja, apenas referenciando a classe. Para constantes numéricas, é comum o uso dos modificadores const e static simultaneamente.
- No caso dos métodos, o modificador static indica que o método é um método de classe, ou seja, seu uso é feito sem um objeto associado, apenas referenciando a classe. O uso de métodos de classe é comum em classes utilitárias, que possuem diversas funções auxiliares que manipulam apenas objetos externos passados como parâmetros. Métodos de classe também são comuns na implementações de métodos que atuam como construtores ou fábricas de objetos.

1.11 Exemplo

- Representação de formas geométricas em uma estrutura de classes

Anotações

Aula 8 - Framework Qt, interface gráfica simples

- Interface Gráfica Simples
- Entrada de Dados na Interface Gráfica

Seja bem vindo a aula sobre o Framework Qt e sua utilização para construção de interfaces gráficas. Veremos em duas partes como utilizar alguns dos mais diversos componentes disponíveis no framework para dar funcionalidade com recursos gráficos nativos a nossas aplicações. No final desta aula esperamos que você esteja preparado para iniciar o desenvolvimento de produtos reais no mercado de software.

1 Interface Gráfica Simples

- Projeto gráfico com Designer
- Rótulos
- Botões
- Sinais e Slots
- Eventos
- QString

Anotações

Nossa jornada na produção de interfaces gráficas com o framework Qt começará pelos seguintes itens:

Projeto Gráfico com o Designer: Conheceremos um pouco da ferramenta visual de criação de janelas e componentes gráficos

Rótulos: Elementos básicos para criação de textos indicativos

Botões: O recurso mais simples de entrada de informação e tomada de decisão pelo usuário

Sinais e Slots: Como conectar os eventos do usuário da interface gráfica com código

Eventos: O tratamento de eventos gerados fora do mecanismo de sinais e slots

QString: As strings do framework Qt, que possuem um pouco mais de funcionalidades do que as strings STL

Exemplo: Um exemplo fechando a unidade e utilizando todos os itens acima

Anotações

1.1 Projeto Gráfico com o Designer

- Qt GUI Application
- Novo esqueleto na função main
- Utilização de Herança + Códigos gerados automaticamente
- Layout feito no Designer (arquivos UI)
- Janela principal herda QMainWindow
- Atributos acessíveis a partir do Qt Designer

Agora que estudaremos janelas e componentes gráficos, vamos precisar utilizar alguns recursos a mais do Qt Creator. A primeira mudança será na criação dos projetos. De agora em diante criaremos projetos do tipo GUI (do inglês para interface gráfica do usuário).

- Os projetos GUI possuirão um esqueleto diferente, principalmente na função main.
- Toda janela é implementada por uma classe através de herança, e seu código será parcialmente gerado pela ferramenta Designer
- Os arquivos do Designer aparecem no editor na seção Forms e tem a extensão UI.
- Nossas aplicações possuem uma janela principal que será derivada do tipo QMainWindow
- A alteração de diversos atributos pode ser feita diretamente no Designer ou no código fonte da classe

Anotações

1.2 Rótulos

- Elemento textual informativo
- “Estáticos” para o usuário
- Podem ser alterados em tempo de execução
- Armazenado como atributo, tipo QLabel
- Formatação extra com folhas de estilo (como HTML/CSS)

Os rótulos, Labels em inglês, são os elementos gráficos mais simples, tipicamente compostos apenas de texto. São tipicamente usados para indicar alguma informação curta, como descrição de um componente de entrada. No Designer o acesso aos rótulos fica disponível no painel lateral com o nome de sua classe associada: a QLabel

- Embora os rótulos sejam estáticos do ponto de vista do usuário, eles podem ser manipulados pelo programador
- Cada rótulo criado através do Designer será representado por um atributo criado na classe que modela a janela, com isso é possível acessar e modificar suas propriedades posteriormente
- Opções de formatação que não estão disponíveis através de métodos podem ser feitas usando folhas de estilo, mas para isso é preciso estudar um pouco da formatação usando atributos CSS (os mesmos usados em páginas Web)

Anotações

Não se esqueça de repetir e modificar os exemplos a vontade, explorando outras possibilidades.

1.3 Botões

- Tipo usado =>QPushButton
- Rótulo interno
- Clicks geram sinais
- Dinâmicos Habilitar/Desabilitar
- Formatação extra com folhas de estilo

A maneira mais simples de tomada de decisão em uma interface gráfica, ou seja, obter uma entrada do usuário é utilizando botões. O Qt possui algumas classes diferentes de botões, mas concentraremos no mais simples que é a QPushButton.

- Os botões possuem um rótulo interno, que pode ser manipulado tal como o QLabel
- O tratamento do click nos botões é feito pelo mecanismo de sinais e slots, que veremos em tópico separado

Anotações

- Como os botões são elementos dinâmicos, temos mais métodos que podem ser usados para habilitar/desabilitar sua utilização e alterar seu estado para o caso de botões de três estados
- Do mesmo jeito que os rótulos, também é possível usar folhas de estilo para mudar a aparência de aspectos visuais que não possuem métodos diretos para alterá-las

1.4 Sinais e Slots

- Componentes emitem sinais para ações do usuário
- Vamos implementar slots para processar esses sinais
- Assinatura Slot = Assinatura Sinal
- connect(emissor, sinal, receptor, slot)
- Declaração do slots em nova seção “public slots:”
- Definição do slot como método normalmente
- Métodos setXyz() dos componentes são slots!
- Conexões simples sinal \Rightarrow slot entre componentes no próprio Designer

Sinais e slots são o mecanismo de geração e tratamento de eventos de interface dentro do framework Qt. Todo componente pode emitir um ou mais tipos de sinais e outros componentes podem tratar esses sinais através de slots.

Anotações

- Não vamos criar componentes novos, então veremos apenas a criação de slots
- O slots deve possuir a assinatura do sinal que pretende tratar, para o caso do sinal emitir parâmetros
- O método connect é usado para ligar sinais a slots, sendo que um mesmo sinal pode estar ligado a vários slots e um mesmo slot pode receber vários sinais de mesma assinatura

O mecanismo de sinais e slots é muito útil do ponto de vista de engenharia de software pois é possível unir os componentes do programa sem alteração no relacionamento entre as classes. Dessa maneira, cada componente fica bem isolado e independente.

- Na declaração da classe, temos uma nova seção que identifica os slots
- Todo slot é um método normal e sua implementação no arquivo CPP se dará normalmente
- Os métodos do tipo set, ou seja, os que fazem alteração nos atributos dos componentes do Qt também são slots, ou seja, é possível conectar sinais diretamente a uma alteração de outro componente sem a necessidade de escrever código intermediário
- Além disso, esse tipo de conexão mais simples de sinais e slots pode ser feito diretamente através do Designer

Anotações

1.5 Eventos

- Útil para ações que não produzem sinais
- Implementados com sobreescrita na herança
- Declaração virtual protected
- Possibilidade de aceitar/recusar os efeitos do evento

O tratamento direto de eventos é uma maneira menos simples e robusta, porém necessária em alguns casos onde não há forma de fazer o mesmo usando sinais e slots.

- O código para tratar um evento deve ser implementado através de herança, fazendo sobreescrita do método correspondente ao evento que se espera tratar.
- Esse método implementado é chamado internamente pelo próprio componente base, algo que só é possível pelo fato do método ser do tipo virtual e protected (para evitar sua utilização fora do contexto da herança).
- Um recurso possível apenas com eventos e não com sinais e slots é a interrupção da propagação do evento, ou seja, impedir que sua consequência natural ocorra

Anotações

1.6 *QString*

- Implementação e utilização parecida com std::string
- Conversão QString::toStdString
- Método .arg() útil para parametrizar texto com valores variáveis

O framework Qt também possui uma implementação própria de strings, que contém todas as funcionalidades das strings STL e ainda vai um pouco além. Dessa forma, de maneira geral não é preciso se preocupar com a utilização, porém duas funcionalidades são muito importantes:

- A primeira é a conversão de QString para string, que pode ser feita pelo método toStdString. Assim, sempre que um texto vindo da interface gráfica precisar ser utilizado por alguma biblioteca da STL, como a fstream por exemplo, o mesmo pode ser convertido facilmente
- A QString possui o método arg, usado para substituir trechos do texto de forma parametrizada e bem mais legível do que usando concatenação

Anotações

1.7 Exemplo

- Geração e exibição de 6 dezenas aleatórias (estilo MegaSena)
- Modelagem da interface
- Criação dos slots
- Lógica do sorte
- Múltiplas janelas

Chegamos ao exemplo desta unidade. Vamos fazer um pequeno programa para sortear 6 dezenas distintas,

- Na primeira parte vamos modelar todos os componentes da interface
- Na segunda parte vamos criar slots para o sorteio
- Na terceira parte vamos implementar a lógica de geração das dezenas
- Na quarta parte vamos implementar o botão de nova janela e o evento para fechar o programa ao fechar qualquer uma das janelas

Anotações

Aula 9 - Framework, Entrada de dados na interface gráfica

- Interface Gráfica Simples
- Entrada de Dados na Interface Gráfica

1 Entrada de Dados na interface gráfica

Agora conhiceremos alguns dos principais widgets, ou componentes gráficos, utilizados para interagir com o usuário e organizar o conteúdo de nossas janelas. Os tópicos serão os seguintes:

- Caixas de Texto
- Caixas de Seleção
- Layouts
- Dialogs
- Menus
- Exemplo

Anotações

Com a utilização básica destes componentes e todo conhecimento acumulado até aqui você estará pronto para desenvolver seus primeiros programas e aprender muito mais.

1.1 Caixas de Texto

- Line Edit
- Text Edit
- Plain Edit

Anotações



- Line Edit
 - Manipulação e acesso
 - Sinais
 - Configuração, restrições

Anotações

No Design temos acesso fácil a três tipos de caixas de texto, com forma e objetivos levemente diferentes:

- Line Edit => O tipo mais simples de caixa de texto, permitindo digitação de apenas uma linha de texto
- Text Edit => Um campo de edição de texto completo, permitindo a edição de verdadeiros documentos (incluindo modificação de fontes, formatação de parágrafos)
- Plain Text Edit => Uma versão mais simples da Text Edit, com foco em texto simples de múltiplas linhas

Como o objetivo é conhecer apenas as funcionalidades básicas, vamos nos concentrar apenas na Line Edit, com as seguintes características:

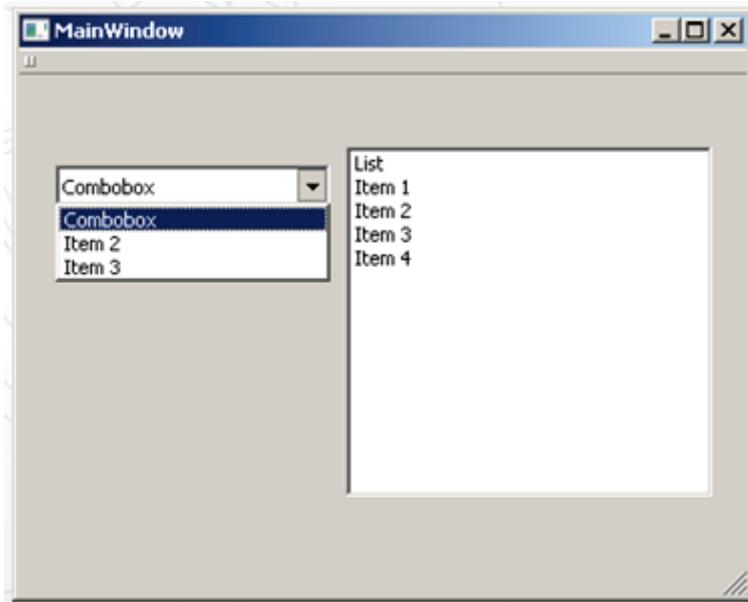
- Métodos para manipular e acessar o texto
- Sinais indicativos de alteração no texto (textChanged, returnPressed)
- Possibilidade de limitar tamanho, restringir possíveis caracteres, modo de digitação de senha

Vamos agora criar duas caixas de texto no Designer e como exercício, vamos conectar o sinal textChanged da primeira com o slot setText na segunda, fazendo com que todo texto digitado na primeira caixa seja copiado para a segunda. Além disso vamos explorar alguns dos atributos para configurar as caixas de texto dentro do Designer.

Anotações

1.2 Caixas de Seleção

- ComboBox
- List



Anotações

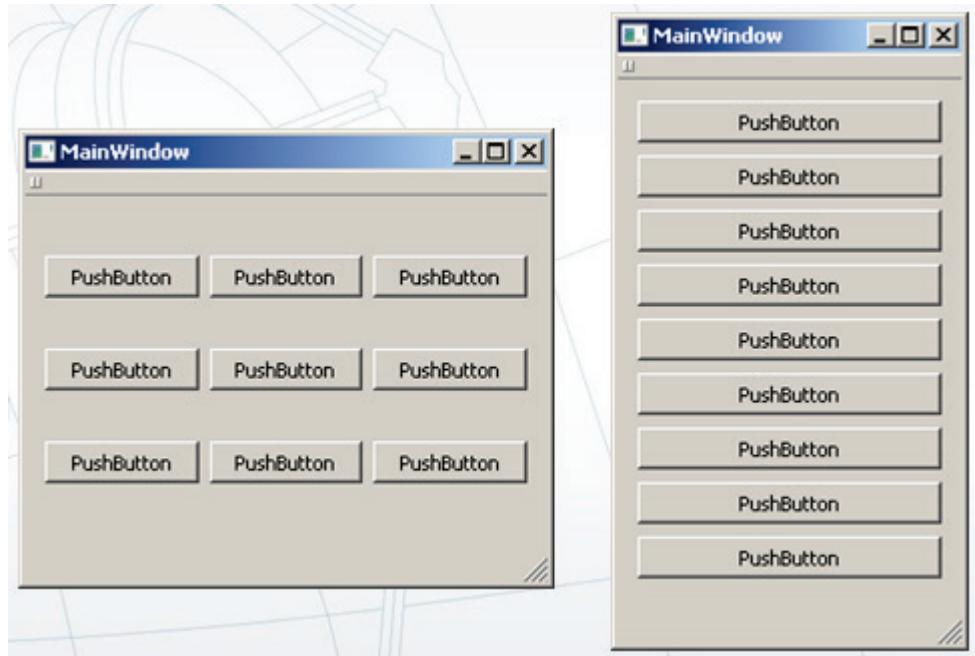
Além de botões e caixas de entrada, um outro recurso muito utilizado são as caixas de seleção. Conheceremos dois tipos:

- A primeira é o Dropdown, que funciona como um botão que, quando clicado, exibe uma lista com múltiplos itens para seleção
- A segunda é o List, que diferente de um dropdown não tem a aparência de botão, mas sim de uma grande caixa de texto com várias linhas, onde podem selecionar uma ou mais delas

1.3 Layouts

- Vertical
- Horizontal
- Form
- Grid

Anotações



Até esse momento posicionamos os widgets nas janelas de forma absoluta, ou seja, determinando exatamente a posição de cada item. Esse tipo de composição da tela não permite que os elementos se ajustem no caso de redimensionamento da janela, para isso precisaremos aplicar layouts e descartar o posicionamento fixo.

Anotações

Toda janela pode ter um layout aplicado e dentro deste layout outros layouts podem ser introduzidos, cada tipo de layout organiza os componentes em seu interior de maneira distinta, sendo eles:

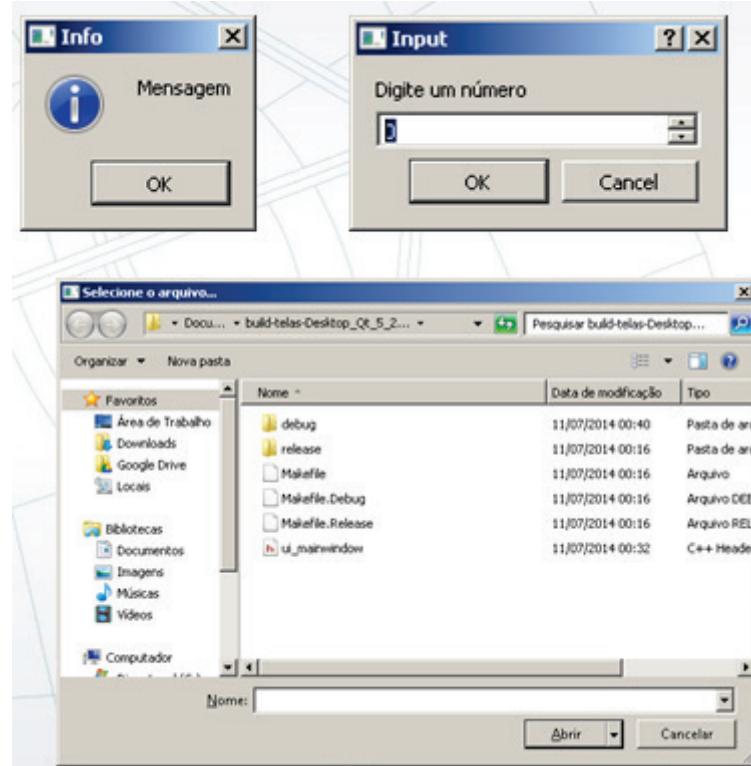
- Vertical layout: Agrupa os componentes um acima do outro
- Horizontal layout: Agrupa os componentes lado a lado
- Form layout: Agrupa os componentes em linhas com 2 colunas (uma para o rótulo e outra para o campo de entrada)
- Grid layout: Distribui os componentes em forma de uma matriz, ou grade de N linhas por M colunas, configuráveis

Além disso temos componentes espaçadores, verticais e horizontais que ajudam a distribuir os componentes quando se deseja criar espaços vazios no layout.

1.4 Dialogs

- QMessageBox
- QLineEdit
- QFileDialog

Anotações



Dialogs são janelas especiais para entrada de dados que bloqueiam o restante da aplicação, por isso são chamadas de modais. Em várias situações esse tipo de comportamento é necessário.

Anotações

Podemos criar dialogs no Designer tal como criamos janelas normais, porém nosso estudo ficará restrito a alguns tipos de dialogs que já vem prontas para o uso no framework Qt. São elas:

- QMessageBox: Dialog que mostra um texto com alguns botões, suas formas principais são para avisos, erros e alertas. Com botões do tipo Sim/Não ou Ok / Cancelar
- QInputDialog: Dialog para entrada de texto, como uma caixa do tipo line edit
- QFileDialog: Dialog para seleção de arquivos e/ou diretórios, usando o mesmo tipo de caixa de seleção visto nos demais programas nativos

1.5 Menus

- Criação de aceleradores com caractere &
- Item de menu ⇒Action ⇒Sinal

A criação de barras de menu com múltiplos níveis no Design é bem simples. E precisamos ficar atento apenas a dois detalhes:

- O caractere & pode ser usado para sinalizar os acelerados, ou seja, a letra que ficará sublinhada no item de menu para facilitar seu acesso por atalhos

Anotações

- Cada item de menu é chamado de action e possui sinais próprios, logo pra cada ação de menu deverá ser feita a conexão com um slot para tratá-la.

1.6 Exemplo

- Agenda telefônica
 - Modelagem
 - Criação de contatos
 - Exclusão
 - Pesquisa

Finalizaremos nosso aprendizado fazendo uma pequena agenda utilizando recursos de interface gráfica. Na primeira parte vamos criar o layout de nossa agenda, com as ações de adicionar e excluir disponíveis através de botões e as opções pesquisar e sair realizadas através de menus.

A segunda parte será a implementação da operação de inclusão, depois na terceira a exclusão e na quarta a pesquisa.

Anotações

Aula 10 - Aperfeiçoamento

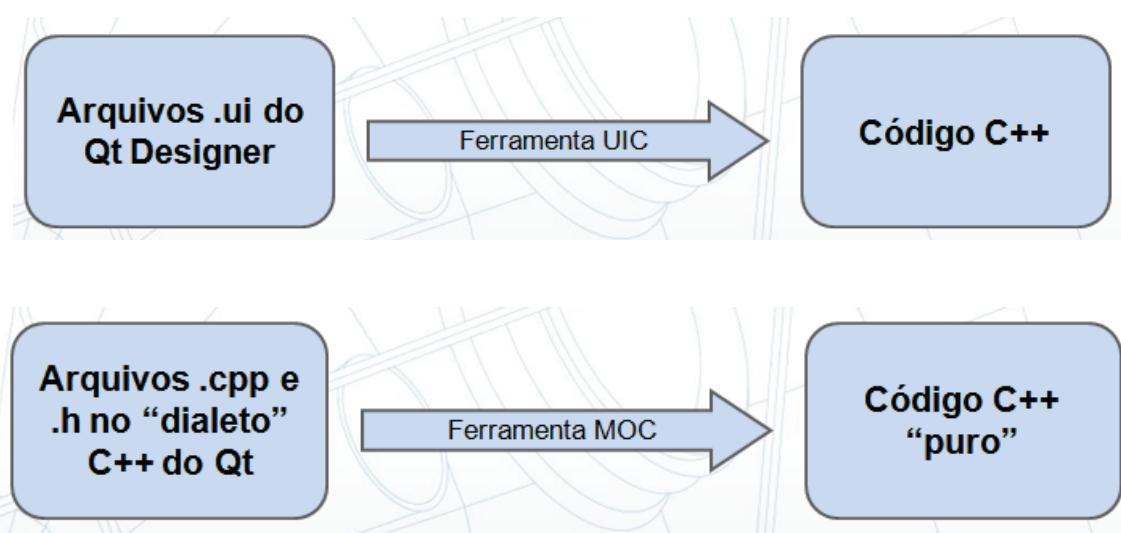
Olá! Seja bem vindo a aula de aperfeiçoamento!

Estudaremos 4 tópicos que permitirão a você conhecer a linguagem C++ um pouco mais profundamente. São eles:

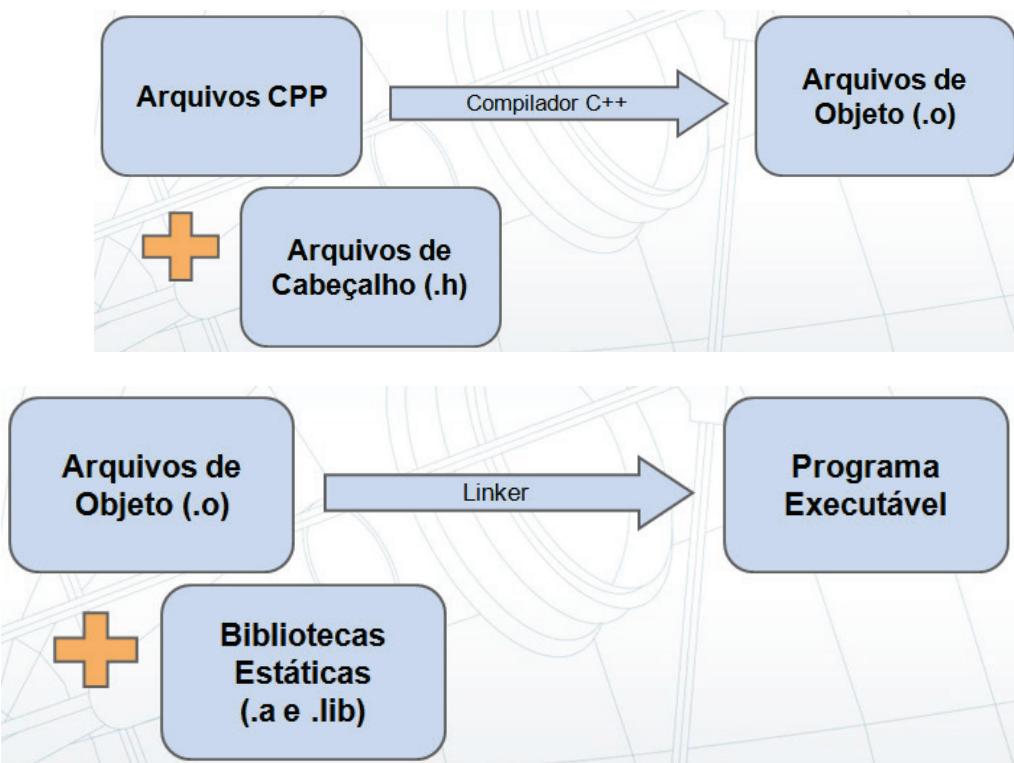
- Compilação e linkagem
- Bibliotecas estáticas, dinâmicas e executáveis
- Macros e Template (Função)
- Tratamento de exceções

Anotações

1 Compilação e linkagem



Anotações



Todo software que escrevemos passa por algum processo de transformação antes de ser executado pelo processador em conjunto com o sistema operacional.

Anotações

No caso da linguagem C++ utilizando o framework Qt, nosso software precisa passar por dois processos, compilação e linkagem. Ambos acontecem automaticamente quando executamos nosso programa através da IDE ou quando utilizamos a comando Build.

No caso particular ainda do Qt, a compilação é dividida em duas partes. O resultado final é o seguinte:

A primeira etapa é a conversão das funcionalidades do Qt para código C++ puro, usando a ferramenta UIC que transforma os arquivos de interface gráfica e após isso usando a ferramenta MOC nos arquivos H e CPP para traduzir o dialeto do Qt (como os sinais e slots) para C++ puro.

A segunda etapa consiste na compilação de cada um dos arquivos de código fonte em um arquivo de objeto executável. O arquivo de objeto já possui instruções diretamente na linguagem do processador, porém ainda não é um programa, apenas um fragmento de um programa.

A última etapa é a linkagem, que consiste em juntar todos os arquivo objetos mais bibliotecas (que também são arquivos objetos) contendo código adicional para executar o programa. O resultado da última etapa será um programa executável ou uma biblioteca, que pode ser estática ou dinâmica.

Anotações

2 Bibliotecas estáticas, dinâmicas e executáveis

- Biblioteca estática
- Código “linkado” ao executável
- Biblioteca dinâmica (DLL)
- Código carregado durante a execução

Além dos executáveis normais, que são programas que podem ser executados diretamente, também podemos criar bibliotecas, que podem ser vistas como um pedacinho de programa reutilizável. Criando bibliotecas podemos separar o desenvolvimento de componentes dos nossos programas que são compartilhados por vários outros aplicativos dentro da nossa empresa.

Temos basicamente dois tipos de bibliotecas:

- As bibliotecas dinâmicas, que no windows tem a extensão DLL
- As bibliotecas estáticas, com a extensão lib ou a (dependendo do compilador)

A diferença entre as duas é que a biblioteca estática tem seu conteúdo copiado para dentro do executável no processo de linkagem. Isso faz a biblioteca dinâmica ser mais rápida em alguns casos e também aumenta显著mente o tamanho final do programa. Outra vantagem é a facilidade de distribuição do programa, isso se dá pelo fato das bibliotecas dinâmicas serem distribuídas em arquivos separados, tendo seu código carregado durante a execução do programa.

Anotações

3 Macros e Template (função)

- Eliminação de código redundante
- Macros: Recurso simples, herdado do C
- Templates: Recurso sofisticado, exclusivo do C++
- Objetivo: Criação de funções genéricas

É comum durante o desenvolvimento de software se deparar com situações onde o programador se vê criando código repetitivo, as vezes com pequenas modificações. Um caso particularmente especial é a criação de funções onde apenas o tipo dos parâmetros é diferente, porém o corpo da função é igual.

Para isso, temos duas ferramentas, os macros herdados diretamente da linguagem C, e os templates criados na linguagem C++. Ambos os recursos podem ser utilizados para diversas coisas, porém vamos nos concentrar apenas no objetivo de criar funções genéricas.

Anotações

4 Tratamento de exceções

- Exceção: Sinalização de uma condição de erro crítica
 - Interrompe o fluxo normal do programa
 - Operação throw
 - Dispara uma exceção
 - Bloco try/catch
 - Permite interceptar e tratar a exceção (executar código alternativo)
 - Exceção não-tratada ⇒ Finaliza o programa

O tratamento de exceção é um recurso que permite a sinalização de erros, ou condições excepcionais, que interrompem o fluxo normal do programa e desviam para um possível fluxo alternativo de tratamento dessa condição.

Métodos e funções podem usar o comando `throw` para disparar uma exceção. Interrompendo imediatamente o código.

Junto a isso, métodos e funções podem ter blocos do tipo try/catch que interceptam exceções disparadas pelos métodos ou funções que foram chamados dentro do bloco try.

Caso uma exceção disparada não tenha um bloco try/catch para tratá-la o programa é finalizado com uma mensagem indicando a exceção não-tratada no console.

Anotações