



Revista Qt

Edição 4 – Março/Abril - 2010

`#include <stdio.h>
int main(void)
{ printf("Hello World!\n");
return 0;
}`
C/C++
Minicurso de
Programação C++
com Qt

**Como Pensar
como um
Cientista da
Computação –
Cap. 2**



Aplicações Híbridas – Desktop + Web
Parte 3



Acesse o canal da Revista Qt no YouTube:

<http://www.youtube.com/user/revistaqt>



Olá caríssimos leitores...

Desta vez a Revista Qt se superou no atraso da publicação. Apesar das constantes conclamações feitas tanto na RQt como no site, não tenho recebido artigos de outras fontes para publicação. A propósito, deixa eu lembrá-los mais uma vez: quem quiser enviar matérias relacionadas a programação em C++, especialmente usando o Qt, é só enviar o texto para o e-mail:

revistaqt@gmail.com

Nesta edição volto às origens, adotando o BrOffice Draw para composição da revista. Continuamos com o nosso Minicurso de Programação C++ com Qt e com a tradução do livro "How to think like a computer scientist" de Allan B. Downey.

A série Qt + PHP também voltou, desta vez com um exemplo de implementação de CRUD usando Qt + PHP + Zend + MySQL.

Um grande abraço a todos

André Luiz de Oliveira Vasconcelos



www.revistaqt.com

Editor

André Luiz de O. Vasconcelos

Charges

Oliver Widder (Geek and Poke)

Colaboradores

Adriano H. Hedler

Rafael Fassi

Thiago Rossener Nogueira

Fórum QtBrasil



Índice



5 – Iniciar

Minicurso de Programação C++ com Qt



16 – Geek & Poke



18 – Versão Brasileira

Segundo capítulo do livro “Como pensar como um cientista da computação”



30 – Laboratório

Aplicações híbridas parte 3 – Qt + PHP

Olá pessoal.

Em nossa primeira aula, vimos como configurar o ambiente para desenvolvimento em C++ com Qt. Fizemos nosso primeiro programa, que não fazia grande coisa, mas nos permitiu o primeiro contato com o compilador.

Nesta segunda aula, veremos variáveis e tipos de dados em C++.

Pra começar: O que é uma variável?

Vamos imaginar que a memória seja como um imenso armário arquivo repleto de gavetas e que nestas gavetas possamos guardar objetos de diversos tipos.

As gavetas deveriam ser compatíveis com o tipo de objeto que pretendemos guardar nelas. A gaveta que armazenará líquidos, por exemplo, deveria ser revestida com plástico ou outro material impermeável. Imagine o que aconteceria se você tentasse guardar água em uma gaveta de madeira...

Como este nosso armário serve pra guardar qualquer coisa, imagino que teríamos nele gavetas refrigeradas, para que pudéssemos guardar alimentos congelados. Teríamos ainda, gavetas com tratamento anti-mofo para guardar roupas, etc.

Além de ter diversos tipos, precisaríamos de diversos tamanhos de gavetas: gavetas pequenas para guardar pequenos objetos, gavetas de tamanho médio para guardar objetos um pouco maiores e gavetas grandes para guardar grandes objetos.

Nessa analogia, cada gaveta ou cada compartimento desse arquivo seria uma variável, cada objeto seria uma determinada informação e cada tipo de objeto (líquido, sólido, alimento, etc.) seria um tipo de informação.

Da mesma forma como colocaríamos indicadores nas gavetas do nosso armário fictício para facilitar a localização de um determinado objeto, as variáveis também precisam de identificadores, de modo que possamos localizar uma informação específica. Estes identificadores, são os nomes das variáveis.

Suponhamos então que você queira guardar o ano do seu nascimento em uma variável, de modo que possa depois recuperar esta informação. O ano de nascimento é um número inteiro (sem casas decimais), portanto vamos informar ao compilador que precisamos de um espaço na memória do computador para armazenar um número inteiro. O processo criar uma variável em C++ é chamado de “declaração de variável”.



fig. 2.1



Vejamos então como seria a declaração de uma variável para armazenar o ano de nascimento:

```
int anoNascimento;
```

fig. 2.2

A instrução mostrada na figura 2.1 declara uma variável chamada **anoNascimento** do tipo **int** (para armazenamento de números inteiros). A palavra reservada **int** (daqui a pouco veremos o que são palavras reservadas) indica o tipo numérico inteiro de dados (vem de **integer**, que significa inteiro em inglês).

Voltando à viagem do armário no começo da aula, isto seria como pegar um papel, escrever nele o nome do objeto que será guardado em uma gaveta, e colar este papel em uma gaveta apropriada para guardar o o tal objeto. Neste ponto a gaveta ainda estaria vazia, porém reservada para um determinado fim.

Voltando para a programação, a declaração de uma variável, como mostrado na figura 2.1, “diz” ao compilador que na execução do programa queremos que seja reservado na memória um espaço com tamanho suficiente para armazenar um número inteiro.

Mas espera um pouco... como é que o compilador “sabe” de quanto espaço ele precisa para guardar um número inteiro? Boa pergunta. Vamos à explicação:

Em C++, existem diferentes tipos de inteiros, que comportam diferentes faixas de valores. O tipo **int**, utilizado na declaração da variável **anoNascimento**, comporta valores que vão de -2.147.483.648 a +2 147 483 647. Isso significa que se tentarmos guardar na variável **anoNascimento** o valor 2.147.483.648, o compilador dará um aviso, indicando que o valor excede o limite de uma variável do tipo **int**.

No caso do nosso exemplo, nem precisaríamos de tanto espaço para guardar o ano de nascimento. Bastaria uma variável capaz de armazenar números com quatro dígitos e positivos.

Existe um tipo de inteiro chamado **short** em C++ que armazena números na faixa de -32768 a 32767. Ainda é muito grande, mas em termos de espaço na memória, o **short** ocupa a metade do **int**. Enquanto o **int** ocupa 4 bytes, o **short** ocupa apenas 2.

Hoje em dia, parece bobagem pensar em economizar 2 bytes, considerando que temos memórias com alguns Gigabytes (1.000.000.000 bytes) nos computadores, mas acreditem, em um passado recente este tipo de economia fazia diferença.

Nesse curso, para facilitar a nossa vida - principalmente a minha :) - utilizaremos sempre o tipo **int** para armazenar números inteiros, sem nos preocuparmos com otimização de memória.

A figura 2.3 mostra uma tabela com os tipos de dados em C++ com seus respectivos tamanhos em bytes e limites. Observe no entanto que estes valores podem variar de acordo com a plataforma (marca e tipo do processador). Não se preocupem em memorizar os tamanhos e limites de cada tipo. Na prática utilizaremos apenas uns três ou quatro desses tipos de dados.



Tipo de dado	Descrição	Tamanho em bytes	Faixa	
			Início	Fim
char	Inteiro com sinal para armazenamento de caracteres	1	-128	127
unsigned char	Inteiro sem sinal para armazenamento de caracteres	1	0	255
signed char	Idem a char	1	-128	127
int	Inteiro com sinal	2	-32.768	32.767
unsigned int	Inteiro sem sinal	2	0	65.535
signed int	Idem a int	2	-32.768	32.767
short int	Idem a int	2	-32.768	32.767
unsigned short int	Idem a unsigned int	2	0	65.535
signed short int	Idem a int	2	-32.768	32.767
long int	Inteiro longo com sinal	4	-2.147.483.648	2.147.483.647
signed long int	Idem a long int	4	-2.147.483.648	2.147.483.647
unsigned long int	Inteiro longo sem sinal	4	0	4.294.967.295
float	Ponto flutuante	4	3,4E-38	3.4E+38
double	Ponto flutuante com precisão dupla	8	1,7E-308	1,7E+308
long double	Ponto flutuante com precisão dupla longo	10	3,4E-4932	3,4E+4932

fig. 2.3



Até aqui, vimos que o tipo de uma variável corresponde ao tipo de dado que ela pode armazenar, assim como o tipo de gaveta dependia do tipo de objeto que queríamos guardar em nosso armário fictício.

A propósito, vocês podem estar se perguntando o porque do nome: **variável**. É simples, chamamos de variáveis porque o conteúdo delas pode variar. Isto significa que podemos substituir o conteúdo de uma variável por outro, assim como poderíamos trocar o objeto guardado em uma das gavetas do armário.

Mas não vimos ainda como colocar conteúdo em uma variável. Este procedimento é chamado de “atribuição de valor” e pode ser feito no momento em que declaramos a variável ou posteriormente.

Ao declararmos a variável **anoNascimento**, poderíamos na mesma instrução, atribuir-lhe um valor. A declaração da variável com a atribuição do meu ano de nascimento ficaria assim:

```
int anoNascimento = 1967;
```

fig. 2.4

A instrução mostrada na figura 2.4 declara a variável **anoNascimento** do tipo **int** (inteiro) contendo o número 1967, que é o ano do meu nascimento e o ano de lançamento da música “Whiter shade of pale” do Procol Harum (fui longe!).

O sinal de igualdade nesta instrução é o operador de atribuição.

A atribuição de valor pode ser feita em um momento posterior ao da declaração de uma variável. No caso da variável **anoNascimento**, poderíamos ter feito a declaração e a atribuição de valor utilizando duas instruções, como mostra a figura 2.5.

```
int anoNascimento;  
anoNascimento = 1967;
```

fig. 2.5

É muito comum que no momento da declaração de uma variável, ainda não saibamos qual o seu conteúdo. No exemplo que vimos aqui, estou criando uma variável para conter o ano do meu nascimento, mas se pretendêssemos armazenar nesta variável o ano do nascimento do usuário do nosso programa? Neste caso, na declaração da variável ainda não sabemos qual será o seu conteúdo. Sabemos apenas que será um número inteiro.

Vamos então ao nosso primeiro programa desta segunda aula. Ele deverá perguntar ao usuário em que ano ele nasceu e armazenar a resposta em uma variável do tipo **int**, chamada **anoNascimento**.

O procedimento para criar o arquivo usando o Qt Creator foi visto na primeira aula, mas vou repetir aqui apenas mais uma vez. Nos próximos programas, vocês “se viram sozinhos”, ok?

Muito bem, turma... carreguem o Qt Creator e selecionem no menu a opção: “**File** → **New File or Project...**”. A combinação de teclas Ctrl + N produz o mesmo resultado.

No diálogo que será aberto pelo Qt Creator, selecionem a opção “C++” do lado esquerdo e “C++ Source File” do lado direito da janela (figura 2.6).

Para concluir cliquem no botão “**Choose...**”.

Lembrando que nesta primeira fase do curso estamos utilizando o Qt Creator apenas como um editor de código fonte. Não o estamos utilizando para criar e gerenciar um projeto em Qt.

Na verdade nem começamos a usar o Qt propriamente dito e nem o faremos por enquanto. Quando todos estiverem seguros com relação à programação em C++ e à programação orientada a objetos, aí sim começaremos a “mexer” com o Qt.

A propósito, para aqueles que quiserem dar uma explorada no Qt Creator, a Revista Qt está disponibilizando vídeos legendados sobre o Qt Creator no YouTube. O endereço do canal da Revista no YouTube é:

<http://www.youtube.com/user/revistaqt>

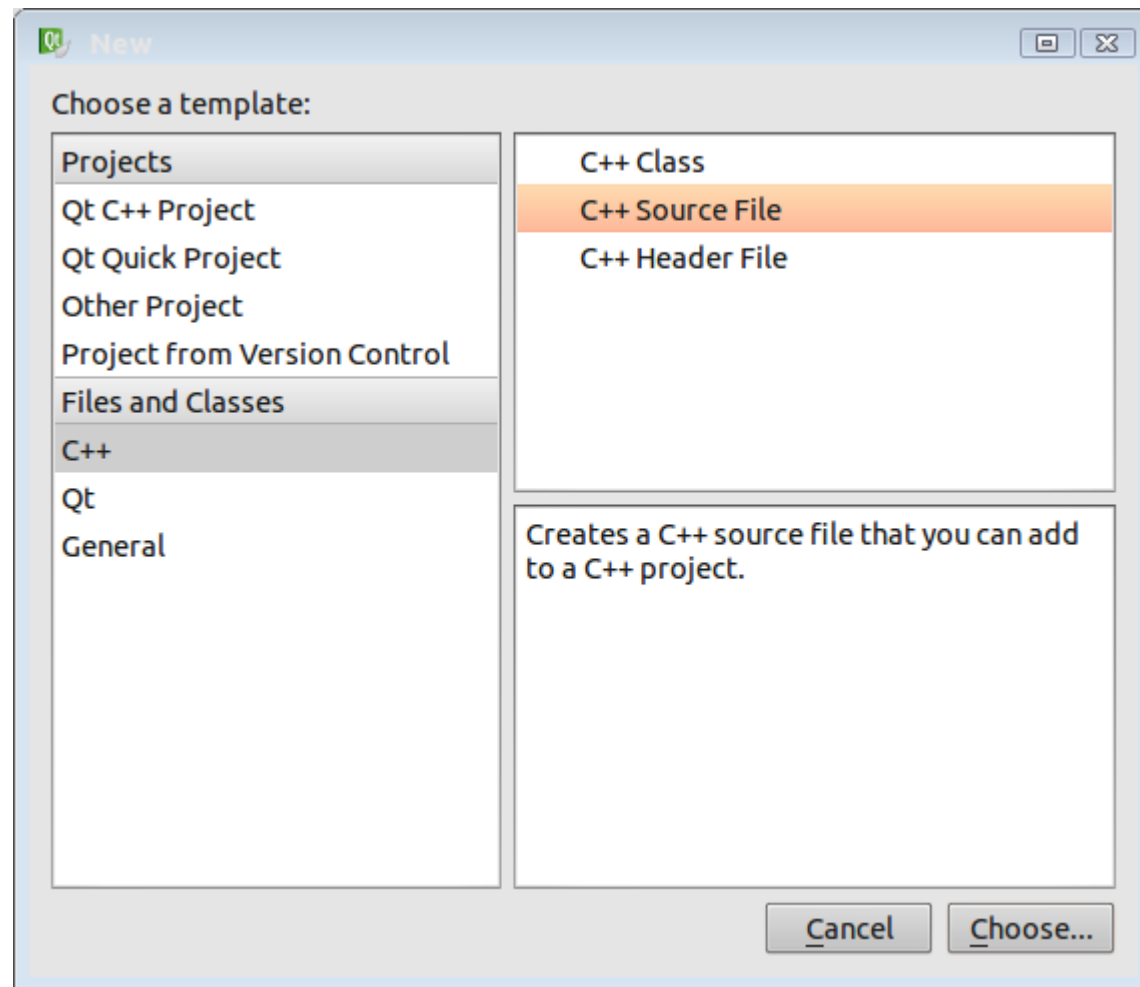


fig. 2.6

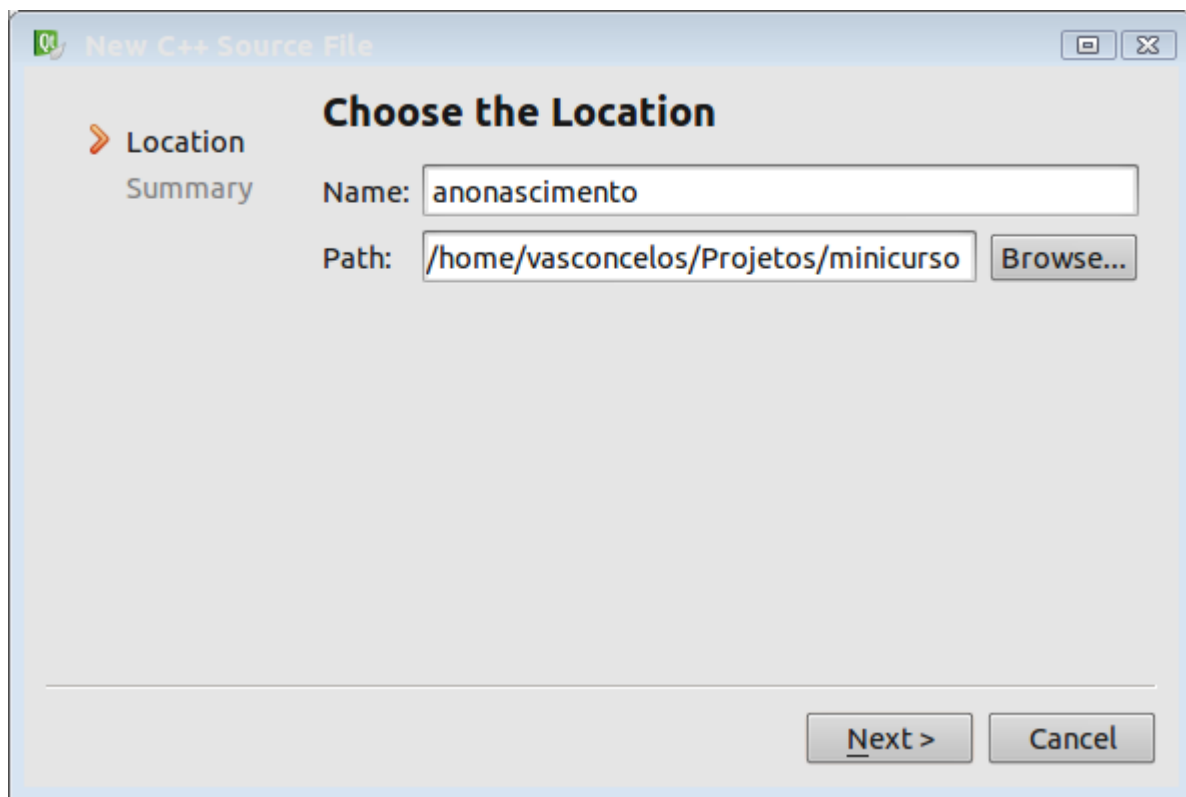


fig. 2.7

No próximo passo, vocês devem informar o nome do arquivo – anonascimento nesse caso – e a localização, ou seja, a pasta onde o arquivo será gerado. No meu caso, salvei o arquivo na pasta:

`/home/vasconcelos/Projetos/minicurso`

Esta pasta é a mesma que utilizei para salvar o arquivo **aula1.cpp**, criado na primeira aula.

Sugiro a todos que façam o mesmo, criando uma pasta para salvar todos os exemplos do curso, para facilitar futuras consultas e também a reutilização de alguns códigos.

Para continuar, cliquem no botão **“Next”** (figura 2.7).

A próxima janela permite selecionar um método para controle de versão. Como não nos interessa no momento, apenas cliquem no botão **“Finish”** (figura 2.8).

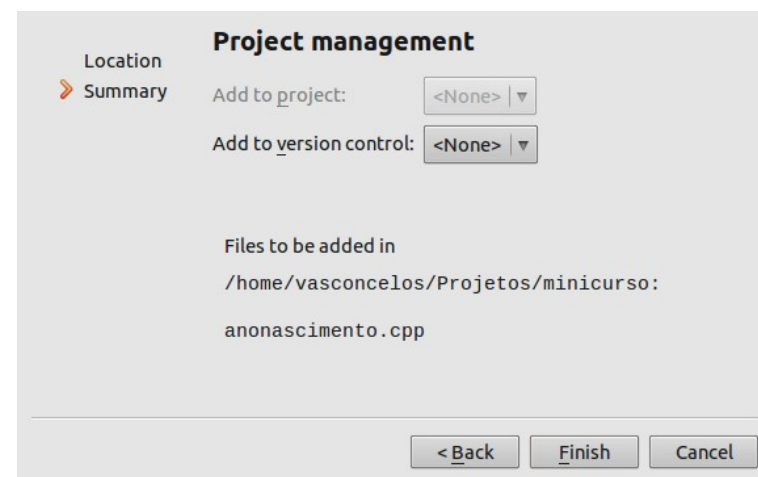
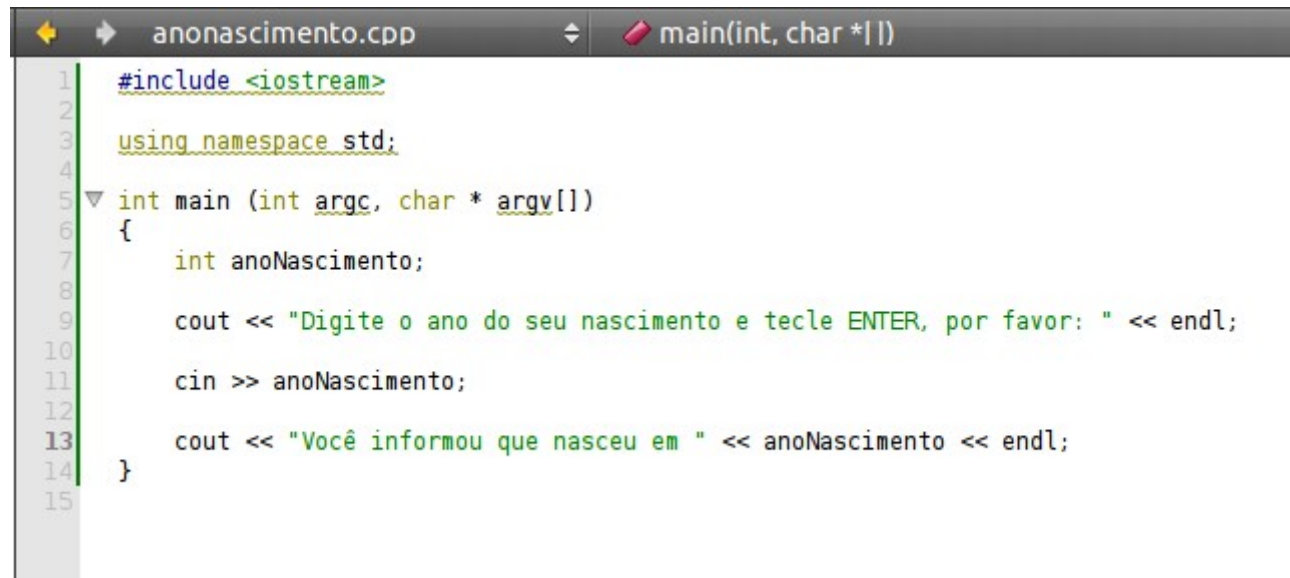


fig. 2.8

A figura 2.9 mostra o código completo do segundo programa desse nosso curso. Em relação ao primeiro, temos como novidade:

- a declaração de uma variável;
- a solicitação de uma informação ao usuário.



```
1  #include <iostream>
2
3  using namespace std;
4
5  int main (int argc, char * argv[])
6  {
7      int anoNascimento;
8
9      cout << "Digite o ano do seu nascimento e tecle ENTER, por favor: " << endl;
10
11     cin >> anoNascimento;
12
13     cout << "Você informou que nasceu em " << anoNascimento << endl;
14 }
15
```

figura 2.9

Bom, agora vamos com muita calma. Este pode ser um programa pequeno e praticamente inútil, mas introduz um importante recurso: a interação com o usuário. Notem que ao invés de colocar um valor fixo na variável, estamos solicitando ao usuário que o faça. A partir de um programa tão simples quanto esse, poderíamos evoluir para um que calculasse a idade do usuário, por exemplo.

Vamos então analisar nosso pequeno programa linha a linha. Recomendo que vocês deixem para digitá-lo depois, quando souberem o que faz cada linha.



As primeiras linhas desse programa são iguais às primeiras linhas do programa que fizemos na aula anterior, mas não custa dar mais uma repassada, então vamos lá:

```
#include <iostream>
```

Esta instrução inclui as definições para os objetos "**cout**" (vimos na primeira aula) e "**cin**" (já veremos para que serve).

```
using namespace std;
```

Esta linha indica ao compilador que estamos utilizando o espaço de nomes chamado "**std**". Não cabe a explicação do que é um espaço de nomes neste ponto do curso. Por ora, basta que saibamos que esta declaração informa ao compilador onde procurar por nomes como "**cin**", "**cout**" e "**endl**". Sem a declaração, teríamos de escrever "**std::cout**" ao invés de simplesmente "**cout**", "**std::cin**" ao invés de "**cin**" e "**std::endl**" no lugar de "**endl**".

Com certeza é bem mais simples declarar o uso do espaço de nomes **std** apenas uma vez no programa do que ter que prefixar cada **cout**, **cin** e **endl** com **std**.

```
int main(int argc, char * argv[])
```

Vimos na primeira aula que a função **main** é o "ponto de partida" dos nossos programas em C++. Em aulas futuras a declaração da função **main** será explicada em mais detalhes.

```
int anoNascimento;
```

Aqui temos a declaração de uma variável do tipo **int** (inteiro) chamada **anoNascimento**. O compilador C++ é *case sensitive*, o que significa que ele faz distinção entre maiúsculas e minúsculas nos identificadores. O nome de uma variável é um identificador. O nome de uma função também é um identificador. Assim, caso vocês escrevam o nome da função **main** deste exemplo, com a letra **m** maiúscula (**Main**), isto causará um erro de compilação. Alguma coisa como: "undefined reference to main", ou seja, o compilador não localizou a função **main**.

O mesmo vale para o nome das variáveis. Como a variável neste programa está sendo declarada com o nome "**anoNascimento**" - com a segunda letra **n** maiúscula, no restante do programa para fazer referência a ela, devemos escrever exatamente como foi escrito na declaração. Desta forma, poderíamos ter duas variáveis chamadas "**anoNascimento**" e "**anonascimento**" em um mesmo programa representando diferentes espaços na memória. Vejam bem... eu disse que poderíamos ter, mas certamente não é recomendável ter duas variáveis com nomes assim tão parecidos porque podemos confundi-las na utilização.

No final desta aula veremos mais sobre os nomes das variáveis.



```
cout << "Digite o ano do seu nascimento e tecle ENTER, por favor: " << endl;
```

Com a variável criada, o próximo passo em nosso programa é imprimir uma mensagem solicitando ao usuário que informe o ano de seu nascimento. Esta instrução é semelhante à que usamos para imprimir a sentença "Funcionou" no programa da primeira aula. O **cout** é um objeto declarado em "iostream" usado para saída de dados para o console (cout = **console output**). O **endl** no final da instrução indica que queremos que seja impressa uma quebra de linha, ou seja a próxima impressão ocorrerá uma linha abaixo da mensagem impressa (endl = **end of line**).

```
cin >> anoNascimento;
```

Agora a última novidade deste programa... o objeto **cin**. Ele é praticamente o inverso de **cout**. Enquanto o primeiro dá saída de informações para o console, este permite a entrada de dados a partir do console (cin = **console input**). A sintaxe (forma como está escrita) desta instrução é muito clara. A entrada obtida a partir do console está sendo armazenada na variável **anoNascimento**. Quando for executada esta instrução fará com que o programa pare e fique aguardando que o usuário digite alguma informação e tecle ENTER. Quando o usuário tecla ENTER, o valor que ele digitou é armazenado na variável **anoNascimento** e a execução prossegue.

```
cout << "Ano do nascimento foi: " << anoNascimento << endl;
```

```
#include <stdio.h>
int main(void)
{
    printf("Hello, World!\\n");
    return 0;
}
```

C/C++

Participe deste projeto.
Envie suas críticas, dúvidas e sugestões para:
revistaqt@gmail.com



Finalmente, o programa imprime uma mensagem mostrando o conteúdo da variável **anoNascimento**, seguido por uma quebra de linha.

A última instrução do nosso programa apenas indica que a função `main` retorna o valor 0. No futuro veremos como utilizar diferentes valores para este retorno.

Pronto. Agora coloquem a mão na massa... podem digitar, compilar e testar o programa. Quem não se lembra como fazer para compilar, deve consultar a primeira aula na segunda edição da RQt.

Experimentem alterar o programa retirando o **endl** no final da primeira instrução `cout`, deixando-a desta forma:

```
cout << "Ano do nascimento: ";
```

Compilem novamente, executem e vejam a diferença.

Vejamos agora quais são as regras para criação de nomes de variáveis em C++.

- Já vimos que o compilador distingue entre maiúsculas e minúsculas nos nomes das variáveis. Assim:

"**nome**" e "**Nome**" são nomes válidos de variáveis diferentes.

- Nomes de variáveis devem começar com letras ou "_" (underline), seguidos de letras, números ou "_". Logo:

"_teste", "Teste_2" e "a" são nomes válidos.

- Nomes de variáveis não podem começar com números. Embora possam conter números como parte de seus nomes. Dessa forma:

"43_anos" e "9meses" são nomes inválidos para variáveis.

"ano2000" e "mes12" são nomes válidos para variáveis

- Nomes de variáveis não podem ser palavras reservadas. Lá no começo desta aula eu mencionei "palavras reservadas" e disse que veríamos depois o que significa. Chegou a hora.

Palavras reservadas são aquelas que possuem para o compilador C++

um significado especial. A palavra **int**, por exemplo é reservada. Se vocês tentarem declarar uma variável chamada `int`, o compilador emitirá uma mensagem de erro, não gerando o executável da aplicação.

A figura 2.10 mostra a lista de palavras reservadas de C++.

asm	auto	break	case
catch	char	class	const
continue	default	delete	do
double	else	enum	extern
float	for	friend	goto
if	inline	int	long
new	operator	private	protected
public	register	return	short
signed	sizeof	static	struct
switch	template	this	throw
try	typedef	union	unsigned
virtual	void	volatile	while

figura 2.10

Para encerrar a aula, alguns comentários sobre nomes de variáveis. É boa prática de programação criar nomes que facilitem a identificação dos dados que as mesmas guardarão.

A variável **anoNascimento**, por exemplo, não nos deixaria em dúvida sobre o seu conteúdo.

Em programas pequenos como o que escrevemos nesta aula, pode não fazer diferença alguma qual o nome da variável, mas acreditem, escreveremos com o tempo programas bem maiores e utilizar uma nomenclatura adequada para as variáveis pode poupar neurônios na hora de ler os códigos.

Experimente substituir o nome da variável "**anoNascimento**" no programa por "**a**". Isso mesmo, só a letra "**a**". O programa vai funcionar da mesma forma que antes, mas no código só identificamos que a variável vai armazenar o ano de nascimento quando olhamos para a linha que solicita ao usuário que informe o seu ano de nascimento.

Em programas pequenos certas recomendações podem parecer meio paranóicas, mas acreditem, à medida em que o tamanho dos códigos aumenta, aumenta a dificuldade de localizar erros.

É isso. Chegamos ao final da nossa segunda aula. Para tirar dúvidas, acessem o grupo do nosso curso:

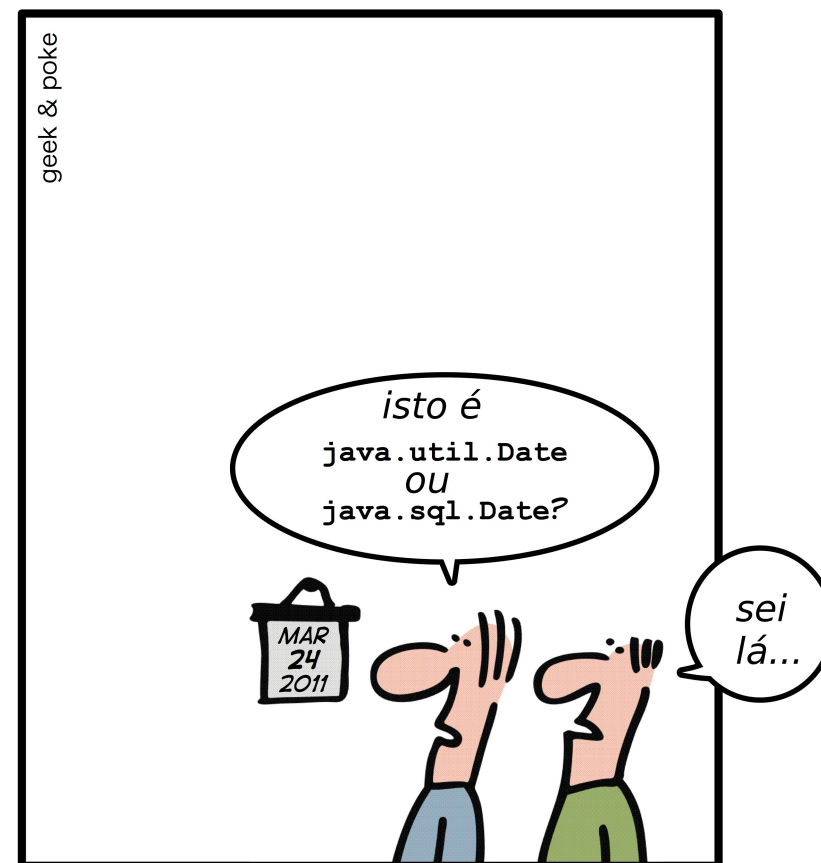
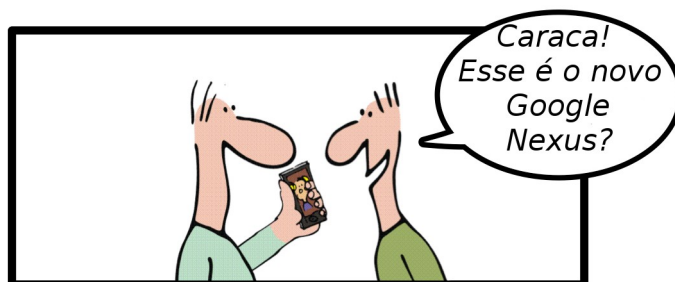
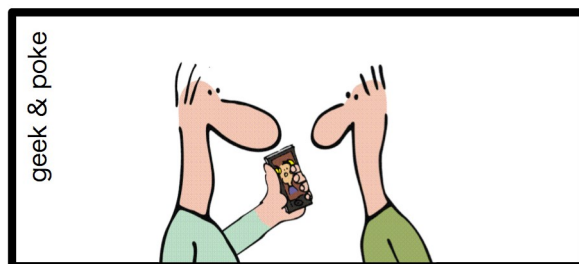
<https://groups.google.com/group/professorvasconcelos?hl=pt-br&pli=1>

Um grande abraço a todos e até mais.



Oliver Widder é o criador do site "Geek and Poke" e colaborador da Revista Qt.

COMO DETECTAR UM GEEK



Programadores odeiam datas



Pesquisa Geek&Poke da semana





Capítulo 2 - Variáveis e tipos

2.1 Mais saídas

Como mencionei no último capítulo, você pode colocar tantas declarações quantas queira na função main. Por exemplo, para mostrar mais de uma linha de saída:

```
#include <iostream.h>
// main: gera algumas saídas simples
void main ()
{
    cout << "Alô, mundo." << endl; // exibe uma linha
    cout << "Como vai você?" << endl; // exibe outra
}
```

Como você pode ver, é legal colocar comentários ao final de uma linha, assim como em uma linha por sí. As sentenças que aparecem entre aspas são chamadas strings (sequência ou cadeia em inglês), porque são formadas de uma sequência de letras. Na verdade, strings podem conter qualquer combinação de letras, números, sinais de pontuação, e outros caracteres especiais.

Frequentemente é útil mostrar a saída de várias declarações de saída todos em uma linha. Você pode fazer isso deixando fora o primeiro endl:

```
void main ()
{
    cout << "Adeus, ";
    cout << "mundo cruel!" << endl;
}
```

Neste caso, a saída aparece em uma única linha como "Adeus, mundo cruel!".

Observe que existe um espaço entre a palavra "Adeus," e o fechamento de aspas. Este espaço aparece na saída, então ele afeta o comportamento do programa.

Espaços que aparecem fora de aspas geralmente não afetam o comportamento do programa. Por exemplo, eu poderia ter escrito:

```
void main ()
{
    cout<<"Adeus, ";
    cout<<"mundo cruel!"<<endl;
}
```

Este programa deve compilar e executar tão bem quanto o original. As quebras de linha não afetam o comportamento do programa também, então eu poderia ter escrito:

```
void main(){cout<<"Adeus, ";cout<<"mundo cruel!"<<endl;}
```

Isto funcionaria também, no entanto você provavelmente percebeu que o programa está ficando mais e mais difícil de ler. Quebras de linhas e espaços são úteis para organizar seu programa visualmente, tornando-o mais fácil ler o programa e localizar erros de sintaxe.

2.2 Valores

Um valor é uma das coisas fundamentais - como uma letra ou um número - que um programa manipula. Os únicos valores que manipulamos até aqui são os valores string que imprimimos, como "Alô, mundo.". Você (e o compilador) podem identificar valores string porque eles estão entre aspas.

Existem outros tipos de valores, incluindo "integers" e "characters". Um integer é um número inteiro como 1 ou 17. Você pode imprimir valores integer da mesma forma que você imprime strings:



```
cout << 17 << endl;
```

Um valor character é uma letra ou dígito ou sinal de pontuação entre aspas simples, como 'a' ou '5'. Você pode imprimir valores character da mesma forma:

```
cout << '}' << endl;
```

Este exemplo imprime uma chave sozinha em uma linha.

É fácil confundir diferentes tipos e valores, como "5", '5' e 5, mas se você prestar atenção à pontuação (aspas duplas e simples), deverá ficar claro que o primeiro é uma string, o segundo é um character e o terceiro um integer. O motivo porque esta distinção é importante deverá ficar clara em breve.

2.3. Variáveis

Uma das mais poderosas características de uma linguagem de programação é a habilidade de manipular variáveis. Uma variável é uma localização nomeada que armazena um valor.

Assim como existem diferentes tipos de valores (integer, character, etc.), existem diferentes tipos de variáveis. Quando cria uma nova variável, você tem que declarar de que tipo ela é. Por exemplo, o tipo character em C++ é chamado char.

A seguinte instrução cria uma nova variável chamada fred que tem o tipo char.

```
char fred;
```

Este tipo de instrução é chamada de declaração.

O tipo de uma variável determina que tipo de valores ela pode armazenar. Uma variável `char` pode conter characters, e não deve ser nenhuma surpresa que variáveis `int` possa armazenar integers.

Existem alguns tipos em C++ que podem armazenar valores strings, mas vamos pular esta parte por enquanto (v. capítulo 7).

Para criar uma variável integer, a sintaxe é:

```
int bob;
```

onde `bob` é um nome arbitrário que você inventa para a variável. Em geral, você irá querer inventar nomes de variáveis que indiquem o que você planeja fazer com a variável. Por exemplo, se você visse estas declarações de variáveis:

```
char firstLetter;  
char lastLetter;  
int hour, minute;
```

você provavelmente teria um bom palpite de que valores seriam armazenados nelas.

Este exemplo também demonstra a sintaxe para declaração de múltiplas variáveis com o mesmo tipo: `hour` e `minute` são ambas inteiros (tipo `int`).

2.4 Atribuição

Agora que criamos algumas variáveis, gostaríamos de armazenar valores nelas.

Fazemos isto com uma declaração de atribuição.



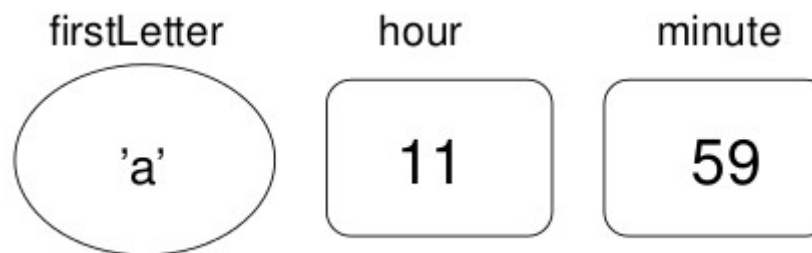
```
firstLetter = 'a'; // dá a firstLetter o valor 'a'  
hour = 11; // atribui o valor 11 a hour  
minute = 59; // "seta" minute para 59
```

Este exemplo mostra três atribuições, e os comentários mostram três diferentes formas como as pessoas algumas vezes falam sobre declarações de atribuição. O vocabulário pode ser confuso aqui, mas a idéia é simples:

- . Quando você declara uma variável, cria uma localização de armazenamento nomeada.
- . Quando faz uma atribuição a uma variável, você dá a ela um valor.

Uma forma comum de representar variáveis no papel é desenhar uma caixa com o nome da variável do lado de fora e o valor da variável dentro.

Este tipo de figura é chamada de diagrama de estado porque mostra em qual estado cada variável está (você pode pensar nisso como o "estado de espírito" da variável). Este diagrama mostra o efeito das três declarações de atribuição:



Algumas vezes, eu uso formas diferentes para indicar diferentes tipos de variáveis. Estas formas devem ajudá-lo a lembrar que uma das regras em C++ é que uma variável deve ter o mesmo tipo do valor que você atribui a ela. Por exemplo, você não pode armazenar uma string em uma variável `int`. A seguinte declaração gera um erro de compilação:

```
int hour;  
hour = "Hello."; // ERRADO!
```

Esta regra é algumas vezes uma fonte de confusão, porque existem muitas formas de você converter valores de um tipo para outro, e C++ algumas vezes convertem coisas automaticamente. Mas por enquanto você deve lembrar-se de que, como regra geral, variáveis e valores têm que ter o mesmo tipo, e falaremos sobre casos especiais mais tarde.

Outra fonte de confusão é que algumas strings parecem com inteiros, mas elas não são. Por exemplo, a string "123", que é composta pelos caracteres 1, 2 e 3, não é a mesma coisa que o número 123. Esta atribuição é ilegal:

```
minute = "59"; // ERRADO!
```

2.5 Imprimindo variáveis

Você pode imprimir o valor de uma variável usando os mesmos comandos que usamos para imprimir valores simples.

```
int hour, minute;  
char colon;  
hour = 11;  
minute = 59;  
colon = ':';  
cout << "A hora corrente é ";  
cout << hour;  
cout << colon;  
cout << minute;  
cout << endl;
```



Este programa cria duas variáveis integers chamadas `hour` e `minute`, e uma variável `character` chamada `colon`. Ele atribui valores para cada uma das variáveis e então usa uma série de declarações de saída para gerar o seguinte:

A hora corrente é 11:59

Quando falamos sobre "imprimir uma variável", queremos dizer imprimir o valor da variável. Para imprimir o nome da variável, você tem que colocá-lo entre aspas.

Por exemplo: `cout << "hour";`

Como já vimos antes, você pode incluir mais de um valor em uma única declaração de saída, o que pode tornar o programa anterior mais conciso:

```
int hour, minute;
char colon;
hour = 11;
minute = 59;
colon = ':';

cout << "A hora corrente é " << hour << colon << minute << endl;
```

Em uma linha, este programa imprime uma string, dois integers, um character e o valor especial `endl`. Impressionante!

2.6 Palavras-chaves

Algumas sessões atrás, eu disse que você pode inventar qualquer nome que desejar para suas variáveis, mas isto não é exatamente verdade. Existem certas palavras que são reservadas em C++ porque elas são usadas pelo compilador para interpretar a estrutura do seu programa, e se você usá-las como nomes de variáveis, ele ficará confuso. Estas palavras, chamadas palavras-chaves, incluem: `int`, `char`, `void`, `endl` e muitas mais. A lista completa de palavras-chaves é incluída no padrão C++, que é a definição oficial da linguagem adotada pela International Organization for Standardization (ISO) em 1º de setembro de 1998. Você pode baixar uma cópia eletrônica em:

<http://www.ansi.org/>

Ao invés de memorizar a lista, eu sugeriria que você aproveitasse uma característica disponível em muitos ambientes de desenvolvimento: destaque de código (code highlighting). À medida em que você digita, diferentes partes do seu programa devem aparecer em diferentes cores. Por exemplo, palavras-chaves podem ser azuis, strings vermelhas e outros códigos pretos. Se você digita o nome de uma variável e ele fica azul, cuidado! Você deve obter um comportamento estranho do compilador.

2.7 Operadores

Operadores são símbolos especiais que são usados para representar computações simples como adição e multiplicação. A maioria dos operadores em C++ faz exatamente o que você esperaria que eles fizessem, porque eles são símbolos matemáticos comuns.

Por exemplo, o operador para adição de dois integers é `+`. Todas as expressões a seguir são legais em C++, com significados mais ou menos óbvios:

`1+1 hour-1 hour*60 + minute minute/60`

Expressões podem conter tanto nomes de variáveis quanto valores inteiros. Em cada caso o nome da variável é substituído pelo seu valor antes de que seja executada a computação.

Adição, subtração e multiplicação todos fazem o que você espera, mas você pode surpreender-se com a divisão. Por exemplo, o seguinte programa:

```
int hour, minute;
hour = 11;
minute = 59;
cout << "Número de minutos desde a meia-noite: ";
cout << hour*60 + minute << endl;
cout << "Fração da hora que passou: ";
cout << minute/60 << endl;
```



geraria a seguinte saída:

Número de minutos desde a meia-noite: 719

Fração da hora que passou:

A primeira linha é o que você espera, mas a segunda linha é singular. O valor da variável minuto é 59, e 59 dividido por 60 é 0.98333, não 0. O motivo para a discrepância é que C++ executa uma divisão inteira.

Quando ambos os operandos são inteiros (operandos são as coisas nas quais os operadores operam), o resultado deverá ser também um inteiro, e por definição, inteiros sempre arredondam para baixo, mesmo em casos como este em que o próximo inteiro está tão próximo.

Uma alternativa possível neste caso é calcular um percentual ao invés de uma fração:

```
cout << "Percentual da hora que passou: ";  
cout << minute*100/60 << endl;
```

O resultado é :

Percentual da hora que passou: 98

Novamente o resultado é arredondado para baixo, mas pelo menos agora a resposta está aproximadamente correta. Para obter um resultado ainda mais preciso, podemos usar um tipo diferente de variável, chamado ponto flutuante, que é capaz de armazenar valores fracionados. Chegaremos a este ponto no próximo capítulo.

2.8 Ordem das operações

Quando mais de um operador aparece em uma expressão, a ordem de avaliação depende das regras de precedência. Uma explicação completa de precedência pode ser complicada, mas apenas para você começar:

. Multiplicação e divisão acontecem antes da adição e subtração. Então $2 * 3 - 1$ resulta em 5, não em 4, e $2 / 3 - 1$ retorna -1, não 1 (lembre-se de que a divisão inteira $2/3$ é igual a 0).

. Se os operadores tem a mesma precedência eles são avaliados da esquerda para a direita. Então na expressão $minute * 100 / 60$, a multiplicação acontece primeiro, resultando $5900 / 60$, o que por sua vez retorna 98. Se as operações tivessem sido feitas da direita para a esquerda, o resultado seria $59 * 1$ o que daria 59, o que está errado.

. Sempre que você quiser sobrepor as regras de precedência (ou você não estiver certo quais sejam elas) você pode usar parênteses. Expressões em parênteses são avaliadas primeiro, então $2 * (3 - 1)$ é igual a 4. Você também pode usar parênteses para tornar uma expressão mais fácil de ser lida, com em $(minute * 100) / 60$, mesmo que isto não altere o resultado.

2.9 Operadores para characters

Curiosamente, as mesmas operações que funcionam em integers também funcionam em characters. Por exemplo,

```
char letter;  
letter = 'a' + 1;  
cout << letter << endl;
```

imprime a letra b. Embora seja sintaticamente legal multiplicar characters, quase nunca é útil fazer isto.

Mais cedo eu disse que você deve atribuir apenas valores integer (int) a variáveis integer e valores character a variáveis character, mas isto não é completamente verdadeiro. Em alguns casos, C++ converte automaticamente entre tipos. Por exemplo, o seguinte é legal:

```
int number;  
number = 'a';  
cout << number << endl;
```



O resultado é 97, que é o número usado internamente por C++ para representar a letra 'a'. Porém, é sempre uma boa idéia tratar characters como characters e integers como integers, e apenas converter de um para o outro se houver um bom motivo.

Conversão automática de tipo é um exemplo de problema comum no projeto de uma linguagem de programação, que é a existência de um conflito entre o formalismo, que é a exigência de que linguagens formais devem ter regras simples com poucas exceções, e a conveniência, que é a exigência de que linguagens de programação sejam fáceis de usar na prática

Mais frequentemente, a conveniência ganha, o que é normalmente bom para programadores experientes, que são poupados do rigoroso, porém pesado formalismo, mas ruim para programadores iniciantes, que muitas vezes surpreendem-se com a complexidade das regras e com o número de exceções. Neste livro, tentei simplificar as coisas enfatizando as regras e omitindo muitas das exceções.

2.10 Composição

Até aqui olhamos para os elementos de uma linguagem de programação – variáveis, expressões e declarações - isoladamente, sem falar sobre como combiná-los.

Uma das mais úteis características de uma linguagem de programação é sua habilidade para pegar pequenos blocos de construção e compô-los. Por exemplo, sabemos como multiplicar inteiros (int) e sabemos como exibir valores de variáveis; acontece que podemos fazer as duas coisas ao mesmo tempo:

```
cout << 17 * 3;
```

Na verdade, eu não poderia dizer "ao mesmo tempo", uma vez que na realidade a multiplicação tem que acontecer antes da impressão, mas o fato é que qualquer expressão, envolvendo números, caracteres e variáveis, pode ser usada dentro de uma declaração de impressão.

Nós já vimos um exemplo:

```
cout << hour*60 + minute << endl;
```



Você também pode colocar expressões arbitrárias do lado direito de uma declaração de atribuição:

```
int percentage;  
percentage = (minute * 100) / 60;
```

Esta habilidade pode não parecer muito impressionante agora, mas veremos outros exemplos onde a composição torna possível expressar computações complexas de forma nítida e concisa.

Aviso: Existem limites para onde você pode usar certas expressões; notadamente, o lado esquerdo de uma expressão de atribuição tem que ser um nome de uma variável, não uma expressão. Isto porque o lado esquerdo indica o local para onde vai o resultado. Expressões não representam locais de armazenamento, apenas valores. Então o seguinte é ilegal: `minute+1 = hour`.

2.11 Glossário

variável: Um local nomeado para armazenamento de valores. Todas as variáveis têm um tipo, que determina que valores podem armazenar.

valor: Uma letra, número, ou outra coisa que possa ser armazenada em uma variável.

tipo: Um conjunto de valores. Os tipos que vimos são inteiros (int em C++) e caracteres (char em C++).

palavra-chave: Uma palavra reservada que é usada pelo compilador analisar programas.

Exemplos que vimos incluem int, void e endl.

statements: Uma linha de código que representa um comando ou uma ação. Até aqui, os statements que vimos são declarações, atribuições e declarações de impressão.

declaração: Um statement que cria uma nova variável e determina seu tipo.

atribuição: Um statement que atribui um valor a uma variável.

expressão: Uma combinação de variáveis, operadores e valores que representam um valor único de resultado. Expressões também têm tipos, determinados pelos seus operadores e operandos.

operador: Um símbolo especial que representa uma computação simples como adição ou multiplicação.

operando: Um dos valores aos quais se aplica um operador.

precedência: A ordem na qual as operações são avaliadas.

composição: A habilidade de combinar expressões simples e statements em statements compostos e expressões para representar computação complexa de modo conciso.



Na primeira edição da Revista Qt, foi publicado um artigo demonstrando o desenvolvimento de uma aplicação híbrida – Desktop + WEB. Na segunda edição, tivemos a segunda parte da série, com um exemplo de aplicação um pouco mais complexa, acessando um banco de dados e utilizando o Zend Framework do lado servidor. Nesta edição, a terceira parte da série Qt + PHP traz um exemplo de aplicação simples, porém completa. Esta aplicação representa o que é comumente chamado de CRUD – Create, Retrieve, Update and Delete, ou seja: Criar, recuperar, atualizar e eliminar (referindo-se a registros em um banco de dados).

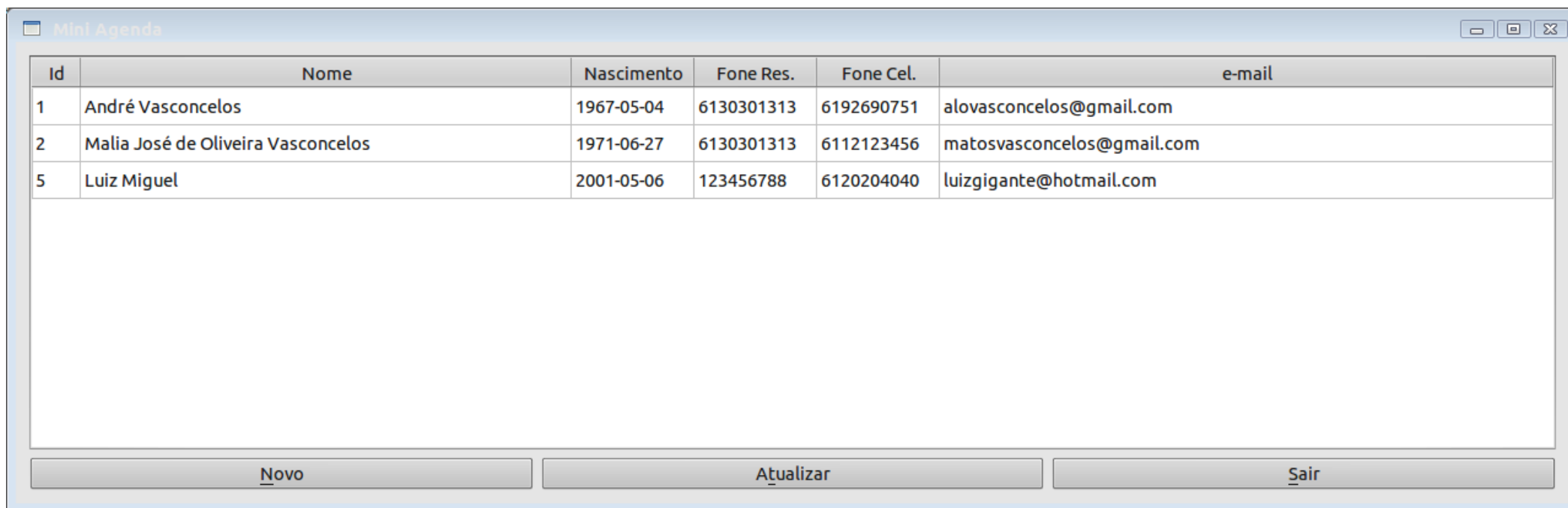


figura 1

A figura 1 mostra como ficará a tela da nossa aplicação, que será uma mini agenda. Para alterar os dados de um contato, devemos clicar duas vezes na linha correspondente ao contato. Neste caso, será apresentada uma outra janela com os dados do contato. Nesta tela (de alteração) teremos um botão “Salvar” para gravar no banco as alterações e um botão “Apagar” que permitirá ao usuário excluir o contato. Para incluir novos contatos, o usuário deve clicar no botão “Novo” mostrado na figura 1. O botão “Atualizar” carrega novamente os dados do banco. O botão “Sair” simplesmente encerra a aplicação.

Como na segunda parte desta série, precisamos dos seguintes ingredientes para fazer esta aplicação:

- ✓ Apache
- ✓ PHP 5
- ✓ MySQL 5
- ✓ Zend Framework minimal

A estrutura da nossa aplicação é a mesma que usamos na aplicação publicada na segunda edição da RQt. No caso da parte servidora da aplicação em PHP utilizei inclusive o mesmo diretório phpapp que usei naquela ocasião.

Recapitulando, na aplicação publicada na segunda edição da RQt criamos um diretório chamado **phpapp** com um subdiretório Classes nele e um link para o diretório library/Zend do Zend Framework. O lado servidor da nossa aplicação naquela ocasião tinha apenas duas classes:

Conexao

Responsável por estabelecer a conexão com o banco de dados da aplicação

ListaEstado

Responsável por recuperar os dados de uma tabela de estados e retornar à aplicação cliente (em Qt).

Para a aplicação que vamos desenvolver agora, faremos algumas mudanças no lado servidor. Para implementar as opções disponíveis agora, criaremos uma classe chamada **Modelo** com os métodos necessários para recuperar, gravar e apagar registros do banco de dados. Esta classe será implementada de modo que suas filhas tenham o mínimo de código necessário. Cada filha da classe Modelo corresponderá a uma tabela no banco de dados. Para a aplicação que estamos criando agora, teremos apenas uma tabela.

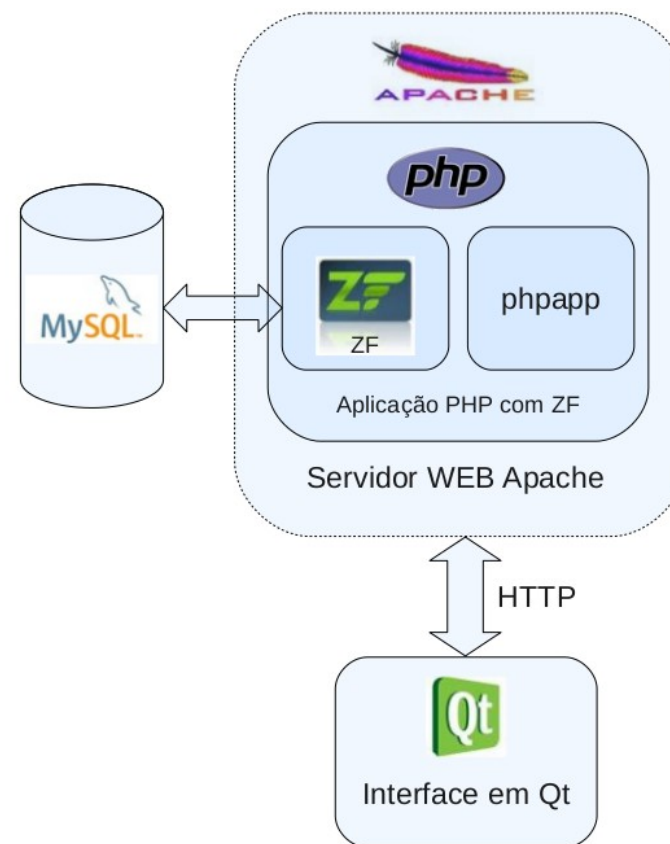
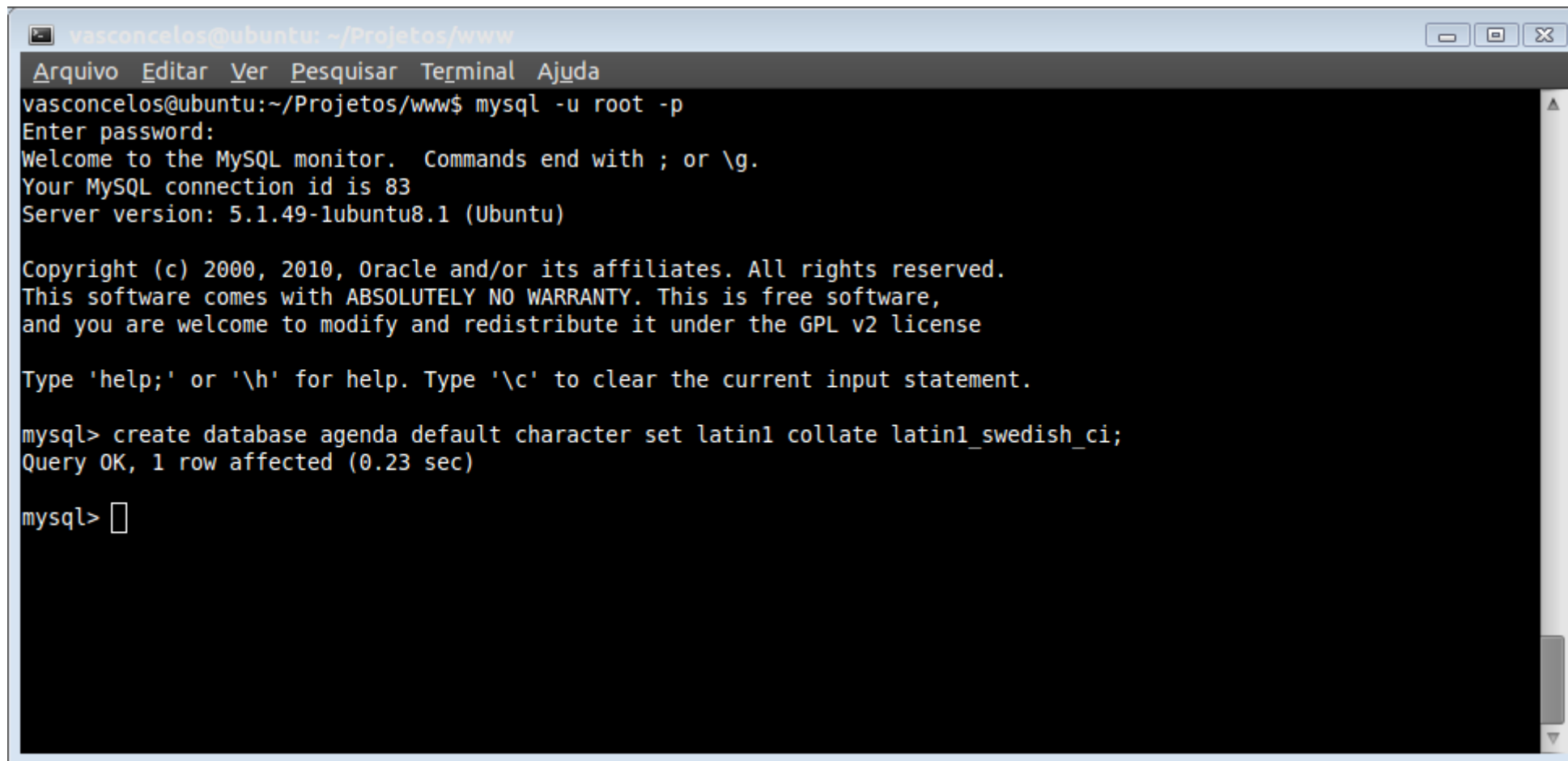


figura 2



Vamos começar pela criação do banco de dados que será usado pela aplicação:

A terminal window titled 'vasconcelos@ubuntu: ~/Projetos/www' with a menu bar (Arquivo, Editar, Ver, Pesquisar, Terminal, Ajuda). The terminal shows the execution of 'mysql -u root -p', followed by a password prompt and a MySQL welcome message. The user then enters the command 'create database agenda default character set latin1 collate latin1_swedish_ci;', which is executed successfully, returning 'Query OK, 1 row affected (0.23 sec)'. The prompt 'mysql>' is shown at the bottom with a cursor.

```
vasconcelos@ubuntu: ~/Projetos/www
Arquivo Editar Ver Pesquisar Terminal Ajuda
vasconcelos@ubuntu:~/Projetos/www$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 83
Server version: 5.1.49-1ubuntu8.1 (Ubuntu)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database agenda default character set latin1 collate latin1_swedish_ci;
Query OK, 1 row affected (0.23 sec)

mysql> 
```

figura 3

A figura 3 mostra o procedimento para conectar no MySql e o comando para a criação do banco.

Nosso próximo passo é criar no banco agenda a tabela **contato**. O comando para criação da tabela é listado a seguir:

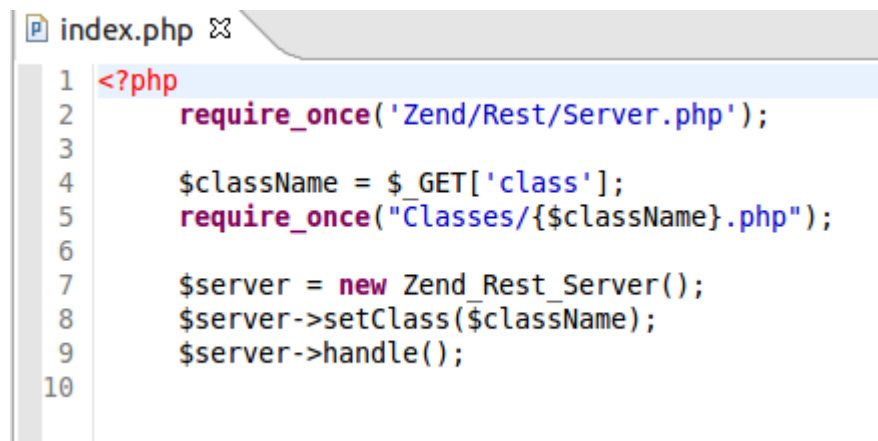
```
CREATE TABLE `contato` (  
  `id_contato` int(11) NOT NULL AUTO_INCREMENT,  
  `nome` varchar(40) NOT NULL,  
  `nascimento` date NOT NULL,  
  `telefone_residencial` varchar(10) NOT NULL,  
  `telefone_celular` varchar(10) NOT NULL,  
  `email` varchar(255) NOT NULL,  
  PRIMARY KEY (`id_contato`),  
  UNIQUE KEY `nome` (`nome`)  
) ENGINE=MyISAM  
  DEFAULT CHARSET=latin1  
  AUTO_INCREMENT=6 ;
```

Com o banco pronto, podemos passar à parte servidora da aplicação, considerando que você tenha o servidor Web Apache e o PHP devidamente configurados. Se você fez a aplicação da segunda edição, então já tem o diretório **phpapp** criado no servidor. Se não tiver, consulte o artigo Qt + PHP da segunda edição da Revista Qt – a partir da página 7 – onde também encontra-se o procedimento para criação do link para o diretório com as bibliotecas do Zend Framework.

Com o servidor Apache/PHP pronto, e o diretório **phpapp** criado, podemos passar às listagens dos programas em PHP que compõem o lado servidor na nossa aplicação. Alguns destes arquivos foram criados no artigo da segunda edição que mencionei antes, mas como tivemos alterações vou reproduzir aqui a listagem de todos eles.

O primeiro da lista é o arquivo index.php, o ponto de entrada para a porção PHP da nossa aplicação. Este é o arquivo chamado pela parte desktop (escrita em Qt) da nossa aplicação para execução de métodos de classes em PHP.

Este arquivo (index.php) deve ser criado no diretório raiz da aplicação web – phpapp. A seguir a listagem do arquivo index.php:



```
1 <?php
2     require_once('Zend/Rest/Server.php');
3
4     $className = $_GET['class'];
5     require_once("Classes/{$className}.php");
6
7     $server = new Zend_Rest_Server();
8     $server->setClass($className);
9     $server->handle();
10
```

figura 4

A figura 4 traz a listagem completa do arquivo index.php. A explicação do que faz cada linha deste arquivo pode ser conferida na página 9 da segunda edição da RQt.

O próximo arquivo que veremos é o Conexao.php que contém a declaração da classe responsável pela conexão e envio de comandos para o banco de dados. Como o código-fonte é maior, o veremos por partes. A figura 5 mostra a inclusão da definição da classe Db do Zend Framework, que usaremos para conexão com o banco de dados, assim como envio de comandos para ele. Este arquivo deve ficar no diretório Classes dentro do diretório da aplicação – phpapp.



```
1 <?php
2 require_once('Zend/Db.php');
3 class Conexao {
4
5     private $db;
6
7     static private $instancia;
8 }
```

figura 5

```
8
9 public function conecta()
10 {
11     if(!is_resource($this->db)){
12         $db = Zend_Db::factory('Pdo_Mysql', array(
13             'host' => 'localhost',
14             'username' => 'root',
15             'password' => 'margrande',
16             'dbname' => 'agenda',
17             'charset' => 'utf8'
18         ));
19         $this->db = $db;
20     }
21 }
22
```

figura 6

A figura 6 traz o código do método conecta, utilizado para estabelecer a conexão com o banco de dados. Observe que neste método é feita uma verificação do atributo db. Caso este atributo seja um recurso, isto indica que a conexão já foi estabelecida. Caso contrário, o método fará a conexão com o banco. Você deve alterar o código neste método, substituindo a senha pela senha do usuário “root” de sua instalação do MySQL.

```
23 static public function getConexao(){
24     if (!isset(self::$instancia)) {
25         $c = __CLASS__;
26         self::$instancia = new $c;
27     }
28     return self::$instancia;
29 }
30
```

figura 7



Na figura 7, continuando com o código fonte do arquivo Conexao.php, temos o método `getConexao`. Este método retorna uma instância do objeto `Conexao`. O método é declarado como estático, porque poderá ser “chamado” sem que haja um objeto da classe `Conexao` instanciado. Esta é uma implementação do *design pattern* conhecido como *Singleton*. Se o atributo **instancia** da classe **Conexao** estiver *setado*, então o método simplesmente retorna este atributo. Caso contrário cria uma nova instância da classe `Conexao` e *seta* o atributo `instancia` com este objeto, antes de retorná-lo (o atributo **instancia**).

```
31 public function camposTabela($tabela)
32 {
33     $this->conecta();
34     return $this->db->describeTable($tabela);
35 }
36
```

figura 8

O próximo método da classe `Conexao` é o `camposTabela` (figura 8), que retorna um array com a coleção de campos de uma tabela passada como argumento para o método. Ele chama o método `conecta` (figura 6) e em seguida através de uma chamada ao método `describeTable` da classe `Zend_Db`, retorna o array com os campos da tabela. A “mágica” para este método ser tão simples está na utilização do Zend Framework.

```
37 public function executaQuery($sql)
38 {
39     $this->conecta();
40     return $this->db->fetchAll($sql);
41 }
42
```

figura 9

A figura 9 mostra o código do método `executaQuery`. Este método recebe como argumento um comando SQL, conecta ao banco de dados (pela chamada ao método `conecta`) e submete o comando ao banco através de uma chamada ao método `fetchAll` da classe `Zend_Db`, retornando o resultado – que no caso será um XML com o retorno da consulta.


```
42  
43 public function executaInsert($tabela, $dados)  
44 {  
45     $this->conecta();  
46     return $this->db->insert($tabela, $dados);  
47 }  
48
```

figura 10

Da mesma forma como temos um método para execução de consultas SQL, temos um método para executar um INSERT em uma dada tabela. Este método, mostrado na figura 10, é o `executaInsert`. O método recebe como argumentos, o nome da tabela e um array com os dados a serem inseridos. Daqui a pouco veremos qual deve ser a estrutura deste array de dados. O método conecta ao banco e solicita a inserção dos dados através de uma chamada ao método `insert` da classe `Zend_Db`.

```
49 public function executaUpdate($tabela, $dados, $condicao)  
50 {  
51     $this->conecta();  
52     return $this->db->update($tabela, $dados, $condicao);  
53 }  
54
```

figura 11

A figura 11 mostra o código do método `executaUpdate` da nossa classe `Conexao`, que como você já deve estar desconfiando, executa comandos UPDATE em tabelas do banco de dados. Além do nome da tabela e do array com os dados para execução do UPDATE, este método recebe ainda um array com as condições para execução do comando. Não se preocupe que mais para a frente veremos a estrutura desse array.



```
55 public function executaDelete($tabela, $condicao)
56 {
57     $this->conecta();
58     return $this->db->delete($tabela, $condicao);
59 }
60
61 }
```

figura 12

Finalmente, pra encerrar o código da classe Conexao (arquivo Conexao.php), temos o método `executaDelete`, mostrado na figura 12, que é o responsável pela execução de comandos DELETE em tabelas do banco. Este método recebe como argumentos o nome da tabela e um array com as condições para a execução do DELETE. A execução do comando se dá pela chamada ao método `delete` da classe `Zend_Db`. A chave fechando na linha 61 corresponde à que foi aberta na linha 3 (v. figura 5), delimitando o código da classe `Conexao`.

Agora que criamos a classe que cuidará da nossa conexão com o banco de dados, vamos à criação da classe que será a responsável pela manipulação de dados correspondente às tabelas do banco. No caso da aplicação que estamos desenvolvendo neste artigo, teremos apenas uma tabela, mas esta classe servirá de base para aplicações maiores no futuro. A seguir, o código da classe `Modelo`, no arquivo `Modelo.php` que deve ficar no diretório `Classes` dentro do diretório da aplicação – `phpapp`.

A figura 13 mostra as primeiras linhas de código da classe `Modelo`. Esta classe possui dois atributos: `$tabela` que armazena o nome da tabela do banco e `$campos` que é um array com os campos que compõem a tabela.

```
index.php  Conexao.php  Modelo.php
1 <?php
2 include_once('Classes/Conexao.php');
3 class Modelo {
4     public $tabela;
5     public $campos;
6 }
```

figura 13

```
6  
7 public function __construct()  
8 {  
9     $this->tabela = strtolower(get_class($this));  
10    $this->campos = array();  
11    $db = Conexao::getConexao();  
12    $this->campos = array_keys($db->camposTabela($this->tabela));  
13    array_shift($this->campos);  
14 }  
15
```

figura 14

A figura 14 apresenta o código do método construtor da classe Modelo. Aqui foram usados alguns “truques” para facilitar ainda mais o desenvolvimento da nossa aplicação servidora.

Pra começar, setamos o atributo **tabela** com o nome da classe convertido para minúsculas. Parece não fazer muito sentido agora, mas nas classes filhas, isto evita que tenhamos que identificar a tabela. Eu explico: na aplicação que estamos desenvolvendo neste artigo, temos uma tabela contato no banco de dados. A classe correspondente a esta tabela se chamará Contato e será filha (extends) da classe Modelo. Dessa forma, o construtor da classe pai irá setar o nome da tabela como “contato”.

Continuando, o construtor inicializa o atributo **campos** como um array, pega da classe Conexao uma conexão com o banco de dados e preenche o atributo **campos** com os campos da tabela, excetuando o primeiro campo (o campo id da tabela).

Prosseguindo com a listagem do arquivo Modelo.php, temos o método **grava**, que será o responsável por gravar os dados nas tabelas dos bancos. Este método é utilizado tanto para inclusão de novos registros como para atualização de registros existentes.

Este método preenche um array com os campos e seus respectivos conteúdos. Claro que para que isto funcione, a aplicação precisa receber pela URL, os nomes corretos dos campos, correspondentes aos nomes dos campos da tabela no banco de dados. Para determinar quando se trata de uma inclusão ou de uma atualização, o método verifica se foi passado um campo **id** pela URL. Se for passado um **id** então trata-se de uma atualização – método `executaUpdate` – caso contrário trata-se de uma inclusão – método `executaInsert`.

Veja o código do método **grava** na figura 15.



```
16 public function grava()  
17 {  
18     $dados = array();  
19     foreach($this->campos as $nome){  
20         if(isset($_GET[$nome])){  
21             $dados[$nome] = $_GET[$nome];  
22         }  
23     }  
24     $db = Conexao::getConexao();  
25     if(!isset($_GET['id'])){  
26         return $db->executaInsert($this->tabela, $dados);  
27     }else{  
28         return $db->executaUpdate($this->tabela, $dados, 'id_'.$this->tabela.' = '.$_GET['id']);  
29     }  
30 }  
31
```

figura 15

Na figura 15, o código do método **apaga** que elimina registros das tabelas no banco de dados. O método deve receber pela URL um campo chamado **id** contendo o id do registro a ser eliminado do banco. O método utilizado para eliminar o registro é o **executaDelete** da classe **Conexao**.

```
32 public function apaga()  
33 {  
34     $db = Conexao::getConexao();  
35     if(isset($_GET['id'])){  
36         return $db->executaDelete($this->tabela, 'id_'.$this->tabela.' = '.$_GET['id']);  
37     }  
38 }  
39
```

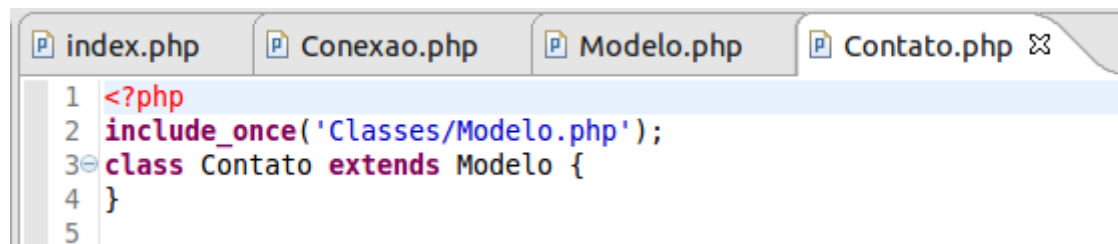
figura 16

A figura 17 traz o último método da classe **Modelo**, o **recuperaTodos** que é responsável pela recuperação de dados das tabelas do banco de dados. Observe que a montagem do comando SQL para recuperação dos dados utiliza o atributo **tabela** que contém o nome da tabela. Caso tenha sido passado pela URL um campo chamado **order**, este será utilizado na cláusula ORDER BY do comando SQL. A última chave fechando na linha 55 refere-se à abertura de chaves feita na linha 3 (v. figura 13).

```
40 public function recuperaTodos()
41 {
42     $db = Conexao::getConexao();
43     $sql = "SELECT
44         *
45         FROM
46             {$this->tabela}";
47     if(isset($_GET['order'])){
48         $sql .= "
49             ORDER BY
50                 {$_GET['order']}";
51     }
52     return $db->executaQuery($sql);
53 }
54
55 }
```

figura 17

Para concluir a parte servidora da nossa aplicação, veremos agora o código da classe Contato, no arquivo Contato.php que deve ser criado no diretório Classes do diretório da nossa aplicação (phpapp). Esta classe será “filha” da classe Modelo. Como a “inteligência” para a manipulação da dados de tabelas foi implementada na classe pai – Modelo – o código da classe Contato será mínimo (como pode-se conferir na figura 18).



```
1 <?php
2 include_once('Classes/Modelo.php');
3 class Contato extends Modelo {
4 }
5
```

figura 18

Isto encerra a parte PHP da nossa aplicação. Acabou ficando muito simples – apenas quatro arquivos – pela utilização do Zend Framework. Agora vem a parte mais divertida da aplicação – seu lado desktop, em Qt.

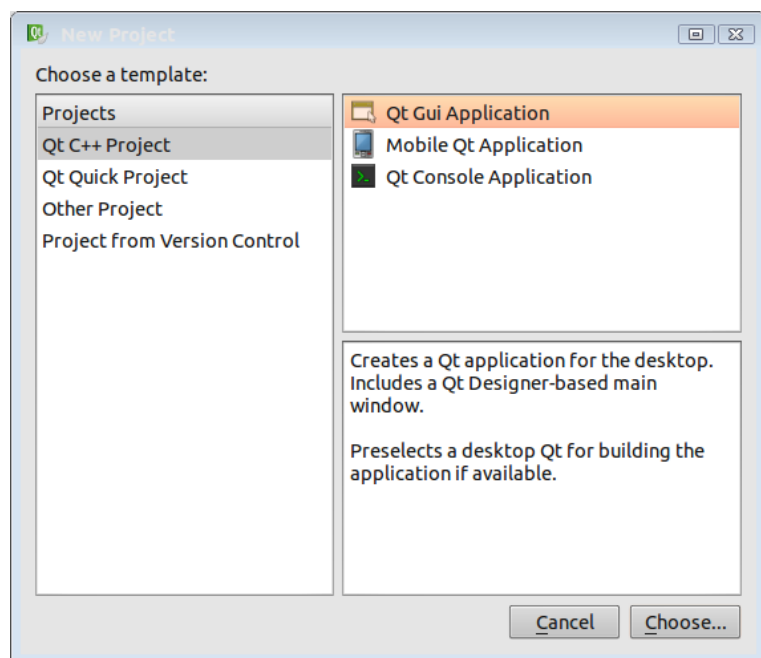


figura 19

No Qt Creator, selecione a opção “Create Project...”, selecionando em seguida “Qt C++ Project” e “Qt Gui Application” (figura 19). Continuando, temos o nome e a localização da aplicação. Usando toda a criatividade que me é peculiar, coloquei o nome da aplicação “Agenda” e selecionei a pasta que utilizo para meus projetos que abusando da mesma criatividade, chama-se Projetos (figura 20).

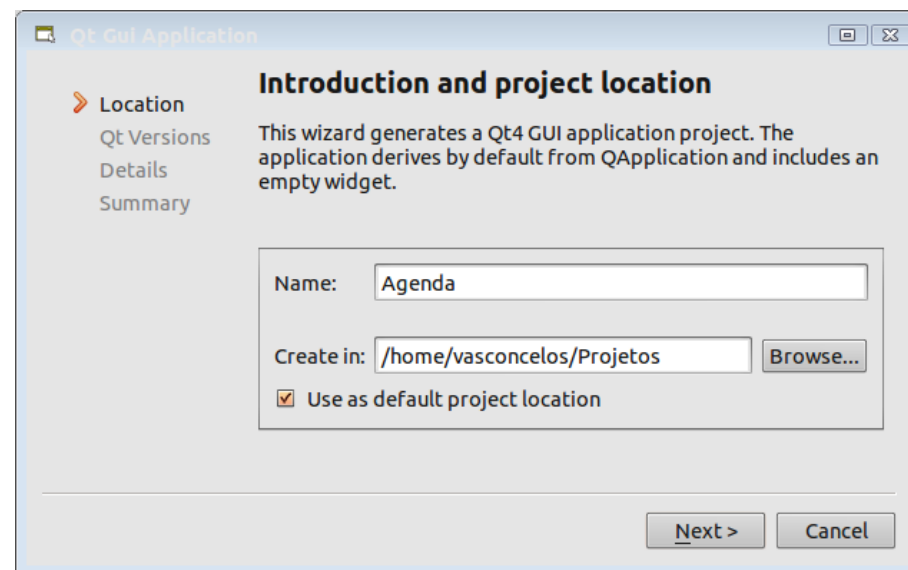


figura 20

Na tela mostrada na figura 21, são apresentadas as versões de Qt disponíveis para o projeto. Normalmente basta clicar no botão Next neste ponto para prosseguir.

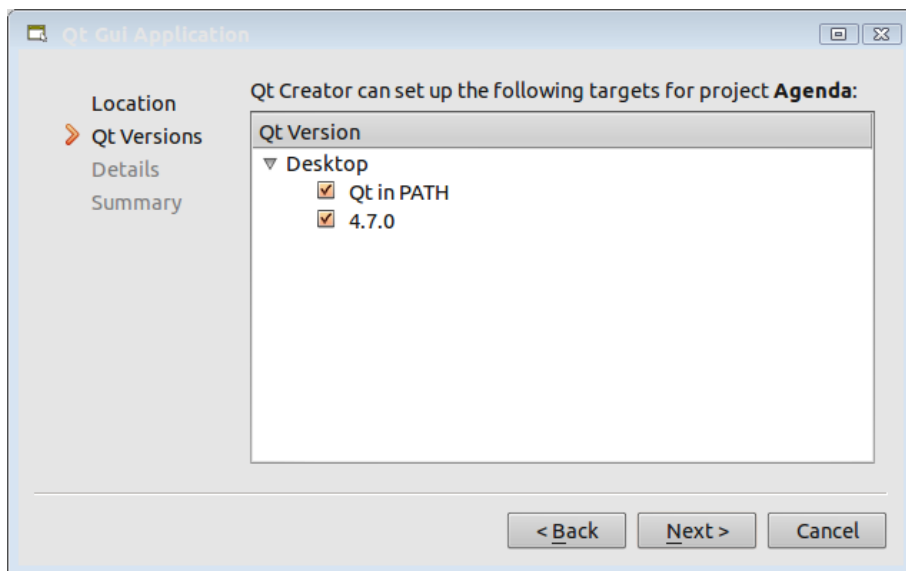


figura 21

Continuando, temos as informações da nossa classe principal do nosso projeto – aquela que corresponderá à janela principal da aplicação. Aceitei a sugestão do Qt Creator como MainWindow, então tudo que precisei fazer foi clicar no botão Next novamente (figura 22).

Na figura 23, o resumo apresentado pelo *wizard* de criação de projetos do Qt Creator. Clicando no botão Finish teremos concluído o processo.

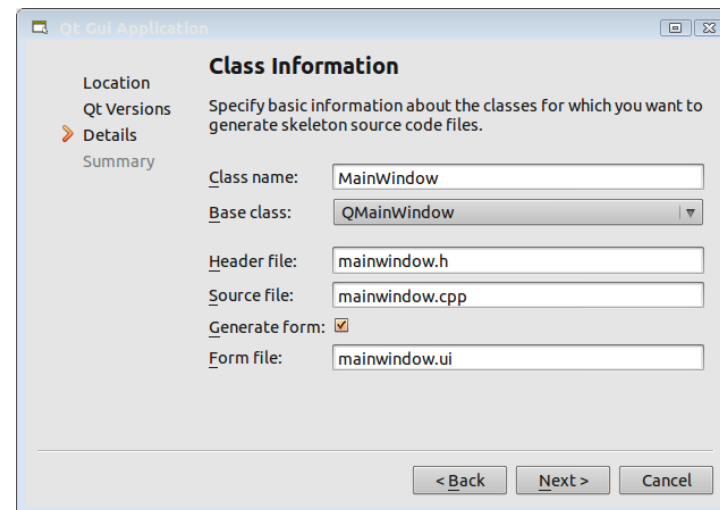


figura 22

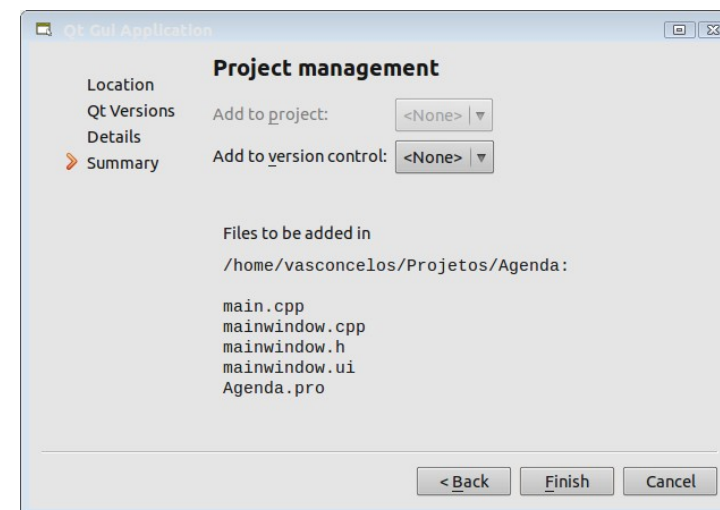


figura 23

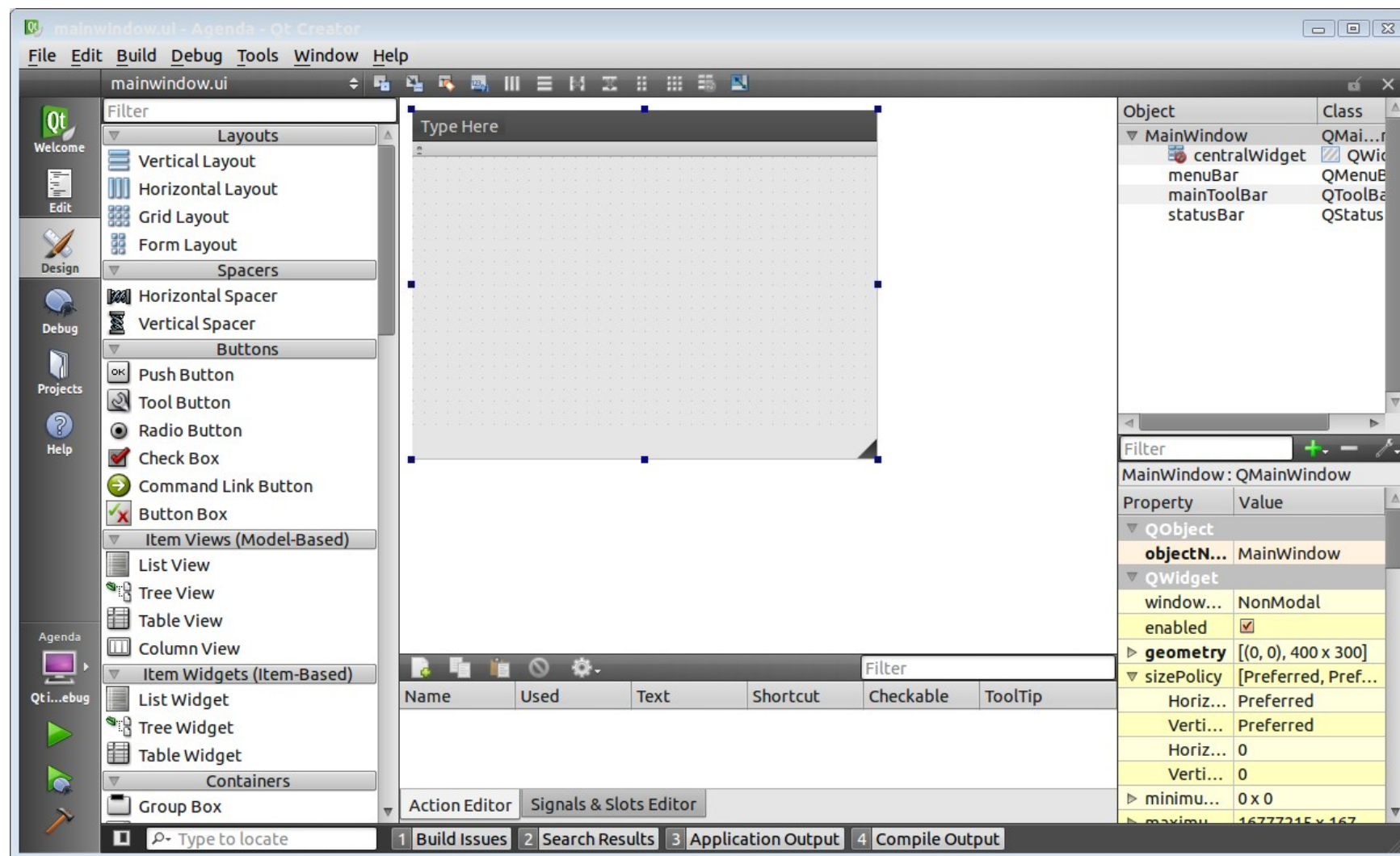


figura 24

Na figura 24, temos a tela do Qt Creator com a janela principal da nossa aplicação. A tela principal da nossa agenda terá um `QTableWidget` que será utilizado para exibir a lista dos dados cadastrados no banco de dados e três `QPushButton`s para os controles da aplicação. Veja na figura 25, a janela principal com os respectivos componentes. Não se preocupe com o posicionamento, pois deixaremos isto por conta do gerenciador de layouts do Qt.

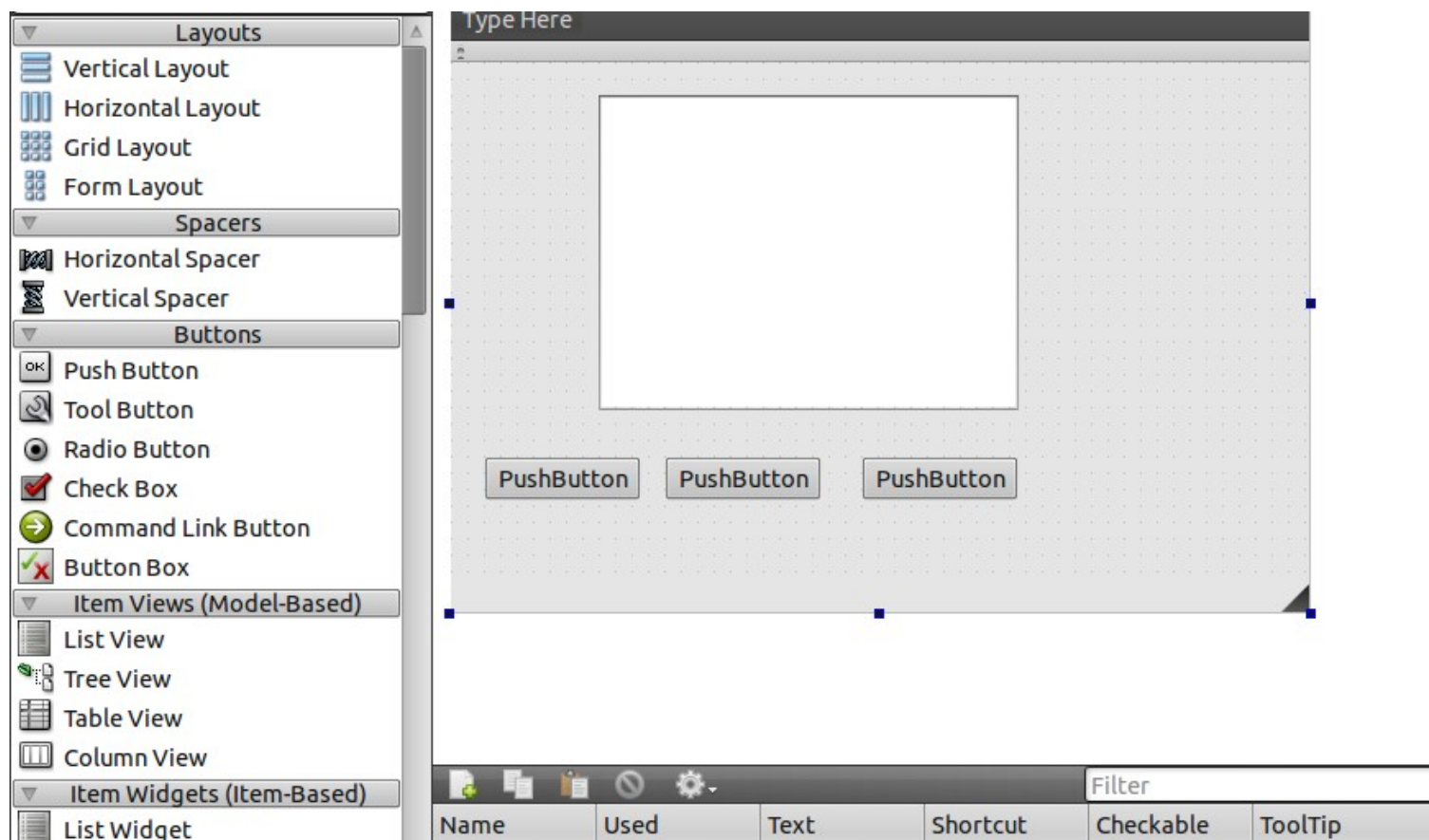


figura 25

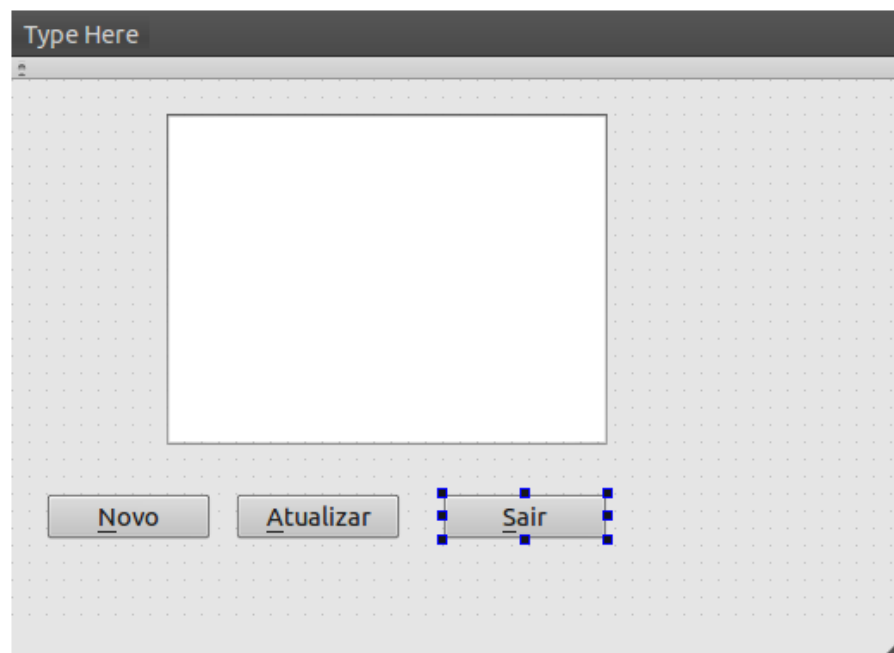


figura 26

Altere os rótulos dos botões conforme mostrado na figura 26. Além dos rótulos dos botões, altere seus nomes conforme mostra a tabela 1.

Rótulo	Nome do componente
&Novo	btnIncluir
&Atualizar	btnAtualizar
&Sair	btnSair

tabela 1

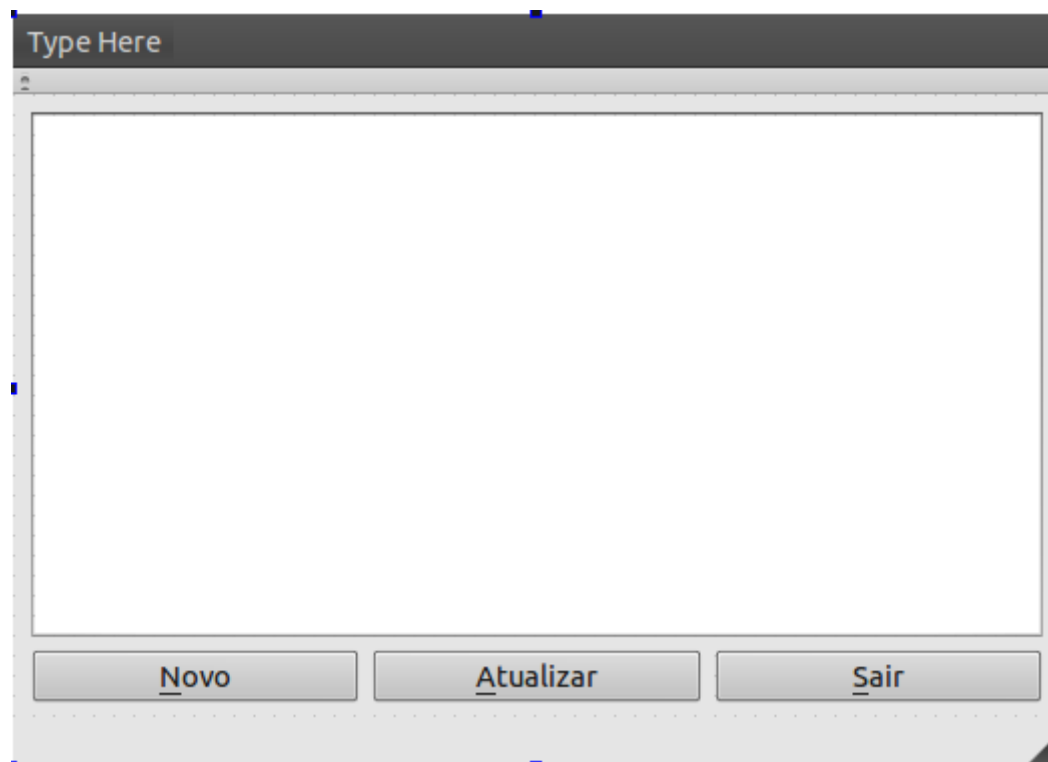


figura 27

Para ajustar o posicionamento dos componentes na tela, clique na janela com o botão direito e selecione a opção **Lay out** em seguida **Lay Out in a Grid**. Outra forma de chegar a este mesmo resultado é clicando na tela para selecioná-la e usando em seguida a combinação de teclas CTRL + G. A aparência da tela deverá ser a apresentada na figura 27.

Com a tela principal da aplicação pronta, vamos à criação da janela que será utilizada para edição dos dados em nossa agenda. Clique no botão Edit no painel do lado esquerdo do Qt Creator para sair do modo de Design. Para criar uma nova janela, clique com o botão direito do mouse na opção **Forms** e em seguida selecione no menu de contexto a opção **Add New**, como mostrado na figura 28.

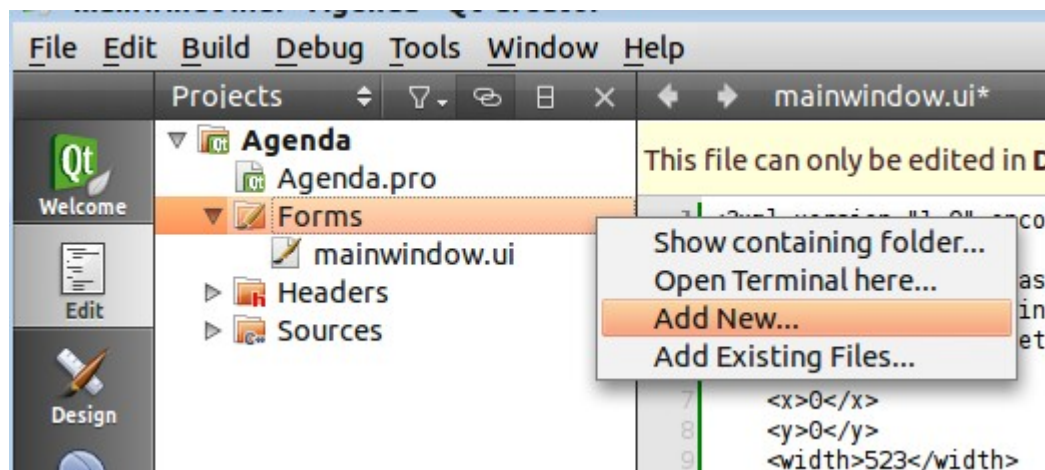


figura 28

Para ajustar o posicionamento dos componentes na tela, clique na janela com o botão direito e selecione a opção **Lay out** em seguida **Lay Out in a Grid**. Outra forma de chegar a este mesmo resultado é clicando na tela para selecioná-la e usando em seguida a combinação de teclas CTRL + G. A aparência da tela deverá ser a apresentada na figura 27.

Com a tela principal da aplicação pronta, vamos à criação da janela que será utilizada para edição dos dados em nossa agenda. Clique no botão Edit no painel do lado esquerdo do Qt Creator para sair do modo de Design. Para criar uma nova janela, clique com o botão direito do mouse na opção **Forms** e em seguida selecione no menu de contexto a opção **Add New**, como mostrado na figura 28.

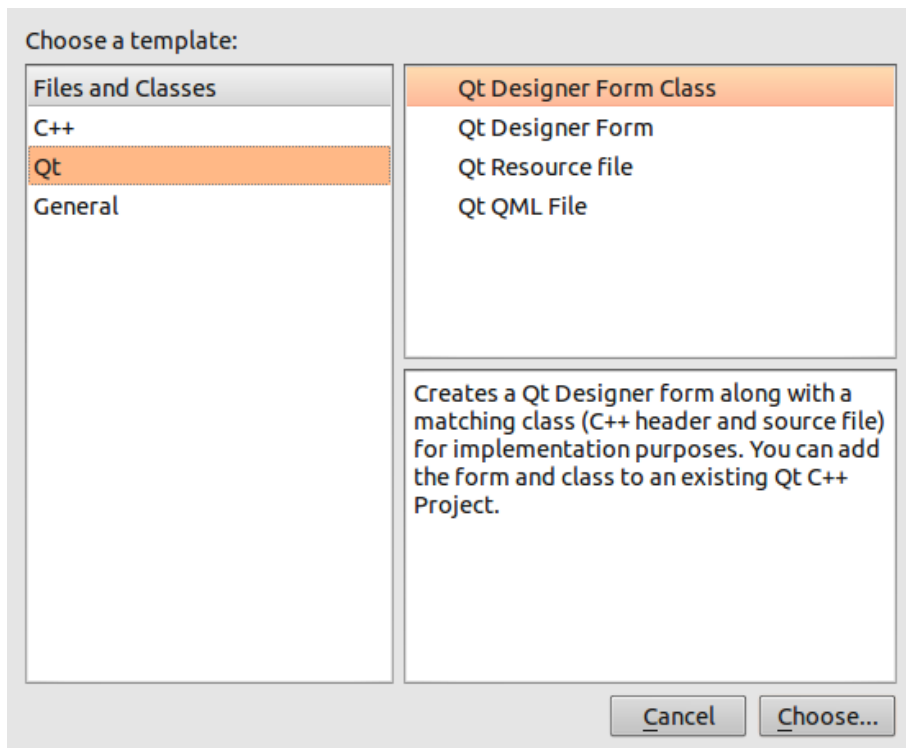


figura 29

Para criação da janela de edição de dados da nossa agenda, selecione as opções **Qt** → **Qt Designer Form Class** (figura 29). Para continuar, clique no botão **Choose**.

A janela de edição de dados da agenda será um QDialog, ou seja, uma janela de diálogo. Acionada a partir da janela principal da aplicação, ela ficará em primeiro plano durante sua exibição. Ao ser fechada, a aplicação habilita novamente a janela principal.

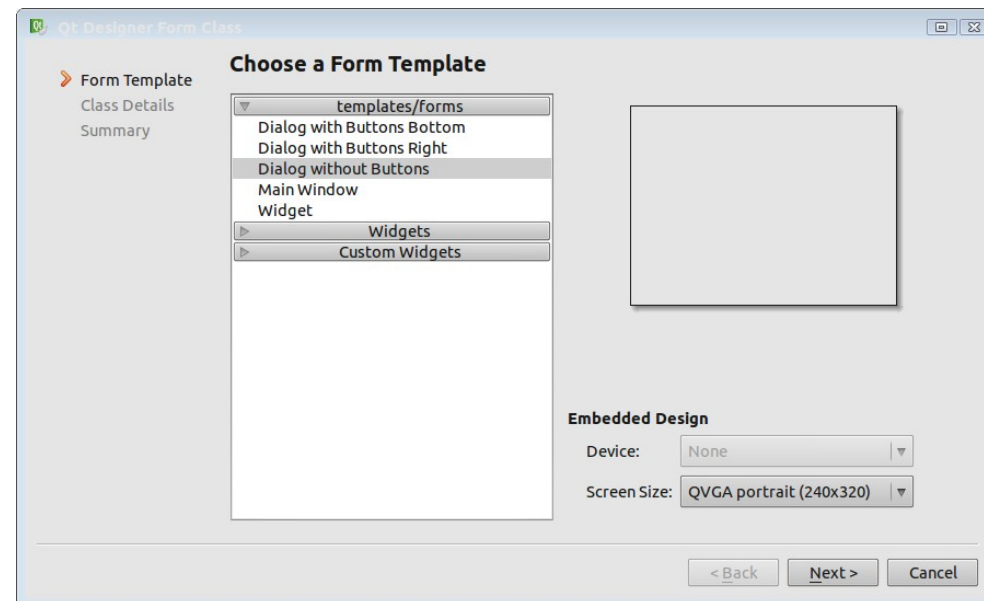


figura 30

Selecione a opção **Dialog without Buttons** e clique no botão **Next** para continuar. Na próxima tela (figura 31), vamos dar um nome para a classe da nossa janela de edição de dados. Criei esta classe com o nome de **DialogoEditar**. Para continuar clique no botão **Next**.

Para concluir o processo, clique no botão **Finish** na tela mostrada na figura 32, que apresenta um resumo da operação de criação do novo diálogo para nossa aplicação.

A exemplo de como fizemos para a janela principal da aplicação, vamos colocar os componentes em nossa janela de edição de dados.

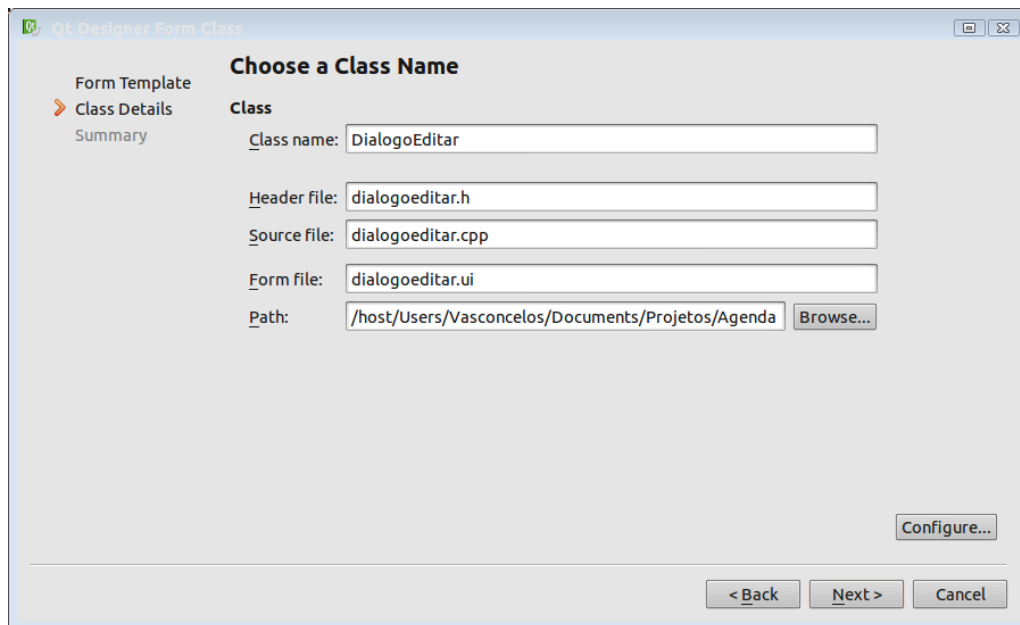


figura 31

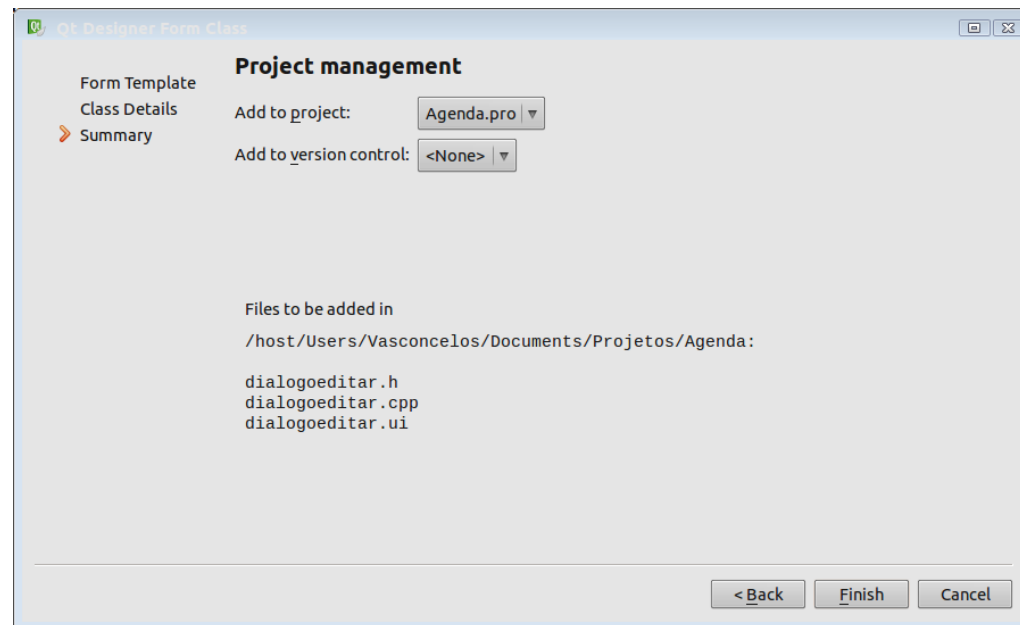


figura 32

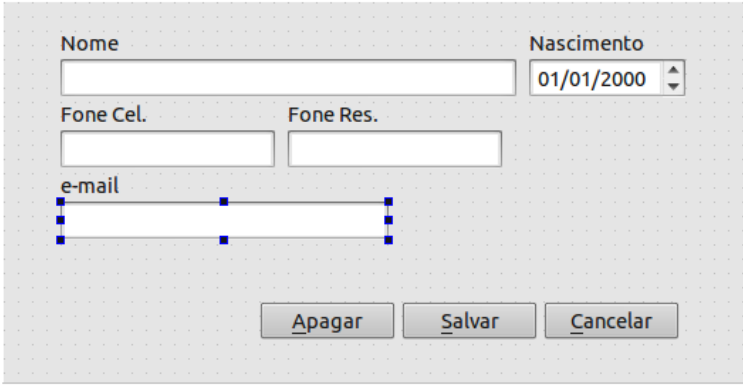
Na janela de edição de dados temos (v. figura 33):

5 componentes QLabel – usados para os rótulos dos campos (Nome, Nascimento, Fone Cel., Fone res. e e-mail.

4 componentes QLineEdit – usados para os campos nome, fone celular, fone residencial e e-mail do contato.

1 componente QDateEdit – usado para o campo data de nascimento do contato

3 componentes QPushButton – usados para acesso às opções: apagar, salvar alterações e fechar a janela.



The image shows a Qt form with a light gray background and a dotted grid. It contains the following elements:

- Nome:** A text input field.
- Nascimento:** A date picker widget showing "01/01/2000".
- Fone Cel.:** A text input field.
- Fone Res.:** A text input field.
- e-mail:** A text input field.
- Buttons:** Three buttons at the bottom: "Apagar", "Salvar", and "Cancelar".

figura 33

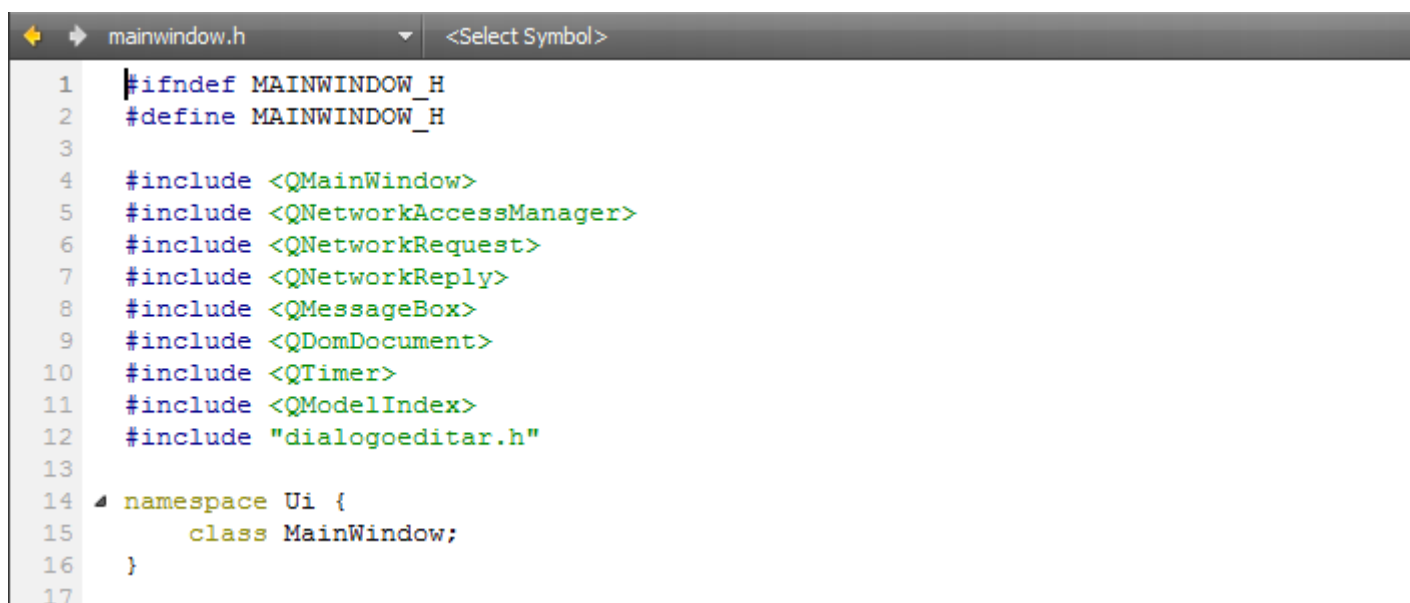
Distribua os componentes no formulário como mostrado na figura 33 ou da forma que achar melhor. Na tabela abaixo estão os nomes de cada um dos componentes.

Campo	Componente	Nome do componente
Nome	QLineEdit	nome
Nascimento	QDateEdit	nascimento
Fone Cel.	QLineEdit	fonecel
Fone Res.	QLineEdit	foneres
e-mail	QLineEdit	email
Botão Apagar	QPushButton	btnApagar
Botão Salvar	QPushButton	btnSalvar
Botão Cancelar	QPushButton	btnCancelar

tabela 2



Com os formulários criados, podemos passar ao código fonte. Vamos começar pela classe MainWindow – a janela principal da nossa aplicação.



```
mainwindow.h
1  #ifndef MAINWINDOW_H
2  #define MAINWINDOW_H
3
4  #include <QMainWindow>
5  #include <QNetworkAccessManager>
6  #include <QNetworkRequest>
7  #include <QNetworkReply>
8  #include <QMessageBox>
9  #include <QDomDocument>
10 #include <QTimer>
11 #include <QModelIndex>
12 #include "dialogoeditar.h"
13
14 namespace Ui {
15     class MainWindow;
16 }
17
```

figura 34

Na figura 34, a primeira parte do arquivo mainwindow.h – a declaração da classe MainWindow. Além dos includes para as classes Qt usadas, observe o include para o arquivo dialogoeditar.h, usada para edição de registros da nossa agenda.

Na figura 35, o restante do código do arquivo mainwindow.h.


```
18 class MainWindow : public QMainWindow
19 {
20     Q_OBJECT
21
22 public:
23     explicit MainWindow(QWidget *parent = 0);
24     ~MainWindow();
25
26 private slots:
27     void recuperarRegistros();
28     void tratarResultado(QNetworkReply * resposta);
29     void gravaDados(bool, QString, QString, QString, QString, QString);
30     void retornoGravaDados(QNetworkReply * resposta);
31     void incluir();
32     void alterar(QModelIndex);
33
34 private:
35     Ui::MainWindow *ui;
36     QNetworkAccessManager * listagem;
37     QNetworkAccessManager * grava;
38 };
39
```

figura 35

Depois do código de declaração da classe MainWindow (no arquivo mainwindow.h), vamos à sua implementação – no arquivo mainwindow.cpp, Como o código deste arquivo é relativamente extenso – pouco mais de duzentas linhas – vamos quebrar o código em partes, apresentadas da figura 36 até a figura 43. Com combinamos que esta série de artigos - Laboratório – seria dedicada ao pessoal com mais conhecimento tanto de programação como de Qt, não veremos a explicação de todos os métodos do programa.



```
mainwindow.cpp  MainWindow::MainWindow(QWidget *)  Line: 58, Col: 19
1  #include "mainwindow.h"
2  #include "ui_mainwindow.h"
3
4
5  MainWindow::MainWindow(QWidget *parent) :
6      QMainWindow(parent),
7      ui(new Ui::MainWindow)
8  {
9      ui->setupUi(this);
10
11      listagem = new QNetworkAccessManager(this);
12      grava     = new QNetworkAccessManager(this);
13
14
15      connect(ui->btnAtualizar,
16              SIGNAL(clicked()),
17              this,
18              SLOT(recuperarRegistros()));
19
20      connect(ui->btnSair,
21              SIGNAL(clicked()),
22              this,
23              SLOT(close()));
24
25      connect(ui->btnIncluir,
26              SIGNAL(clicked()),
27              this,
28              SLOT(incluir()));
```

figura 36

```
29
30 connect(listagem,
31         SIGNAL(finished(QNetworkReply*)),
32         this,
33         SLOT(tratarResultado(QNetworkReply*)));
34
35 connect(grava,
36         SIGNAL(finished(QNetworkReply*)),
37         this,
38         SLOT(retornoGravaDados(QNetworkReply*)));
39
40 connect(ui->tableWidget, SIGNAL(doubleClicked(QModelIndex)), this, SLOT(alterar(QModelIndex)));
41
42 // Configura o QTableWidgetItem
43 ui->tableWidget->setEditTriggers(QAbstractItemView::NoEditTriggers);
44 ui->tableWidget->setSelectionMode(QAbstractItemView::SingleSelection);
45 ui->tableWidget->setColumnCount(6);
46 ui->tableWidget->verticalHeader()->setVisible(false);
47 ui->tableWidget->setColumnWidth(0, 40); // Id
48 ui->tableWidget->setColumnWidth(1, 400); // Nome
49 ui->tableWidget->setColumnWidth(2, 100); // Nascimento
50 ui->tableWidget->setColumnWidth(3, 100); // Fone res.
51 ui->tableWidget->setColumnWidth(4, 100); // Fone cel.
52 ui->tableWidget->setColumnWidth(5, 500); // email
53
54 // Monta o cabeçalho da tabela
55 QStringList cabecalho;
56 cabecalho << "Id" << "Nome"
57             << "Nascimento" << "Fone Res."
58             << "Fone Cel." << "e-mail";
59 ui->tableWidget->setHorizontalHeaderLabels(cabecalho);
```

figura 37



```
60
61     // Chama o SLOT recuperarRegistros();
62     QTimer::singleShot(0, this, SLOT(recuperarRegistros()));
63
64     this->showMaximized();
65 }
66
67 MainWindow::~MainWindow()
68 {
69     delete ui;
70 }
71
72 void MainWindow::recuperarRegistros()
73 {
74     ui->tableWidget->setRowCount(0);
75     ui->tableWidget->clearContents();
76
77     // Monta a URL para requisição dos dados ao servidor
78     QString url = "http://localhost/phpapp/?class=Contato&method=recuperaTodos";
79
80     // Executa a requisição ao servidor
81     listagem->get(QNetworkRequest(QUrl(url)));
82 }
83
84 void MainWindow::tratarResultado(QNetworkReply * resposta)
85 {
86     // Verifica se houve erro na resposta
87     if(resposta->error() != QNetworkReply::NoError){
88         QMessageBox::critical(this, "Erro",
89                               "Não foi possível recuperar dados do servidor");
90         return;
91     }
```

figura 38

```
92
93 // Cria objeto DOM para tratamento do XML de resposta e
94 // verifica se a resposta pode ser atribuída ao este objeto
95 QDomDocument doc;
96 if(!doc.setContent(resposta)){
97     QMessageBox::critical(this,"Erro","Erro tratando resultado");
98     return;
99 }
100
101 // Verifica se o retorno foi gerado pela classe ListaEstado
102 QDomElement classe = doc.documentElement();
103 if(classe.tagName() != "Contato"){
104     QMessageBox::critical(this,"Erro",
105         "O XML recebido não é da classe Contato");
106     return;
107 }
108
109 // Nó correspondente ao nome do método - retorna
110 QDomNode metodo = classe.firstChild();
111
112 // Nó correspondente ao registro
113 QDomNode registro = metodo.firstChild();
114
115 // Nó correspondente ao status - o último
116 QDomNode status = metodo.lastChild();
117 if(status.toElement().text() != "success"){
118     QMessageBox::critical(this,"Erro","O servidor retornou erro");
119     return;
120 }
```

figura 39



```
121
122     // Percorre os registros
123     int linha    = 0;
124     while(!registro.isNull() && registro != status){
125         ui->tableWidget->insertRow(linha);
126         \Users\Vasconcelos\Documents\Projetos\Qt\Agenda.original\#Forms;Child();
127         int coluna    = 0;
128         // Percorre os campos
129         while(!campo.isNull()){
130             QTableWidgetItem * item = new QTableWidgetItem(campo.toElement().text());
131             ui->tableWidget->setItem(linha, coluna, item);
132             coluna++;
133             campo = campo.nextSibling();
134         }
135         linha++;
136         registro = registro.nextSibling();
137     }
138 }
139
140 void MainWindow::incluir()
141 {
142     QModelIndex modelo;
143     alterar(modelo);
144 }
```

figura 40

```
145
146 void MainWindow::alterar(QModelIndex index)
147 {
148
149     DialogoEditar ed(this);
150     if(index.flags() != Qt::NoItemFlags){
151         ed.id = ui->tableWidget->item(index.row(), 0)->text();
152         ed.nome = ui->tableWidget->item(index.row(), 1)->text();
153         ed.nascimento = ui->tableWidget->item(index.row(), 2)->text().remove("-");
154         ed.foneres = ui->tableWidget->item(index.row(), 3)->text();
155         ed.fonecel = ui->tableWidget->item(index.row(), 4)->text();
156         ed.email = ui->tableWidget->item(index.row(), 5)->text();
157         ed.init();
158     }
159     if(ed.exec() == QDialog::Accepted){
160         gravaDados(ed.flagApagar,
161                 ed.id,
162                 ed.nome,
163                 ed.nascimento,
164                 ed.foneres,
165                 ed.fonecel,
166                 ed.email);
167     }
168 }
169
```

figura 41



```
170 void MainWindow::gravaDados(bool flagApagar,
171                             QString id,
172                             QString nome,
173                             QString nascimento,
174                             QString foneres,
175                             QString fonecel,
176                             QString email)
177 {
178     // Monta a URL para requisição dos dados ao servidor
179     QString url = "http://localhost/phpapp/?class=Contato&method=";
180     if(flagApagar){
181         url.append("apaga");
182         url.append("&id="+id);
183     }else{
184         url.append("grava");
185         if(!id.isEmpty()){
186             url.append("&id="+id);
187         }
188         url.append("&nome="+nome);
189         url.append("&nascimento="+nascimento);
190         url.append("&telefone_residencial="+foneres);
191         url.append("&telefone_celular="+fonecel);
192         url.append("&email="+email);
193     }
194     // Executa a requisição ao servidor
195     grava->get(QNetworkRequest(QUrl(url)));
196 }
197
```

figura 42


```
198 void MainWindow::retornoGravaDados(QNetworkReply * resposta)
199 {
200     // Verifica se houve erro na resposta
201     if(resposta->error() != QNetworkReply::NoError){
202         QMessageBox::critical(this, "Erro",
203                               "Erro na gravação do registro");
204         return;
205     }
206     recuperarRegistros();
207 }
```

figura 43



Participe deste projeto. Envie suas críticas, sugestões, dúvidas, artigos para:

revistaqt@gmail.com



```
dialogoeditar.h  <Select Symbol>  Line: 1, Col: 1
1  #ifndef DIALOGOEDITAR_H
2  #define DIALOGOEDITAR_H
3
4  #include <QDialog>
5
6  namespace Ui {
7      class DialogoEditor;
8  }
9
10 class DialogoEditor : public QDialog
11 {
12     Q_OBJECT
13
14 public:
15     explicit DialogoEditor(QWidget *parent = 0);
16     ~DialogoEditor();
17     void init();
18
19     bool flagApagar;
20     QString id;
21     QString nome;
22     QString nascimento;
23     QString fonerres;
24     QString fonecel;
25     QString email;
26
27 private:
28     Ui::DialogoEditor *ui;
29
30 private slots:
31     void gravar();
32     void apagar();
33 };
34
35 #endif // DIALOGOEDITAR_H
36
```

figura 44



```
dialogoeditar.cpp  <Select Symbol>  Line: 1, Col: 1

1  #include "dialogoeditar.h"
2  #include "ui_dialogoeditar.h"
3
4  DialogoEditor::DialogoEditor(QWidget *parent) :
5      QDialog(parent),
6      ui(new Ui::DialogoEditor)
7  {
8      ui->setupUi(this);
9
10     connect(ui->btnSalvar,
11             SIGNAL(clicked()),
12             this,
13             SLOT(gravar()));
14     connect(ui->btnApagar,
15             SIGNAL(clicked()),
16             this,
17             SLOT(apagar()));
18     connect(ui->btnCancelar,
19             SIGNAL(clicked()),
20             this,
21             SLOT(close()));
22     this->flagApagar = false;
23     ui->btnApagar->setVisible(false);
24 }
25
26 void DialogoEditor::init()
27 {
28     ui->nome->setText(this->nome);
29     ui->nascimento->setDate(QDate(this->nascimento.mid(0, 4).toInt(),
30                                 this->nascimento.mid(4, 2).toInt(),
31                                 this->nascimento.mid(6, 2).toInt()));
32     ui->foneres->setText(this->foneres);
33     ui->fonecel->setText(this->fonecel);
34     ui->email->setText(this->email);
35     ui->btnApagar->setVisible(true);
36 }
```

figura 45



```
37
38  DialogoEditar::~DialogoEditar()
39  {
40      delete ui;
41  }
42
43  void DialogoEditar::gravar()
44  {
45      QString dataNascimento;
46      dataNascimento.append(QString::number(ui->nascimento->date().year()));
47      dataNascimento.append("-");
48      dataNascimento.append(QString::number(ui->nascimento->date().month()));
49      dataNascimento.append("-");
50      dataNascimento.append(QString::number(ui->nascimento->date().day()));
51      this->nome = ui->nome->text();
52      this->nascimento = dataNascimento;
53      this->foneres = ui->foneres->text();
54      this->fonecel = ui->fonecel->text();
55      this->email = ui->email->text();
56      QDialog::accept();
57  }
58
59  void DialogoEditar::apagar()
60  {
61      this->flagApagar = true;
62      QDialog::accept();
63  }
```

figura 46

É isso. Execute a aplicação e teste suas funcionalidades, incluindo, excluindo e alterando registros na agenda. A figura 47 mostra a aparência da nossa pequena Agenda. Como esta aplicação foi concebida para servir como exemplo, tentei simplificá-la ao máximo. Não implementei uma opção de pesquisa, por exemplo. A propósito este pode ser um bom exercício pra você: criar um diálogo que solicite o dado a ser localizado na agenda, procure no banco e exiba os registros localizados.

Nas próximas edições da RQt veremos outros exemplos de aplicações híbridas como esta. Até lá.

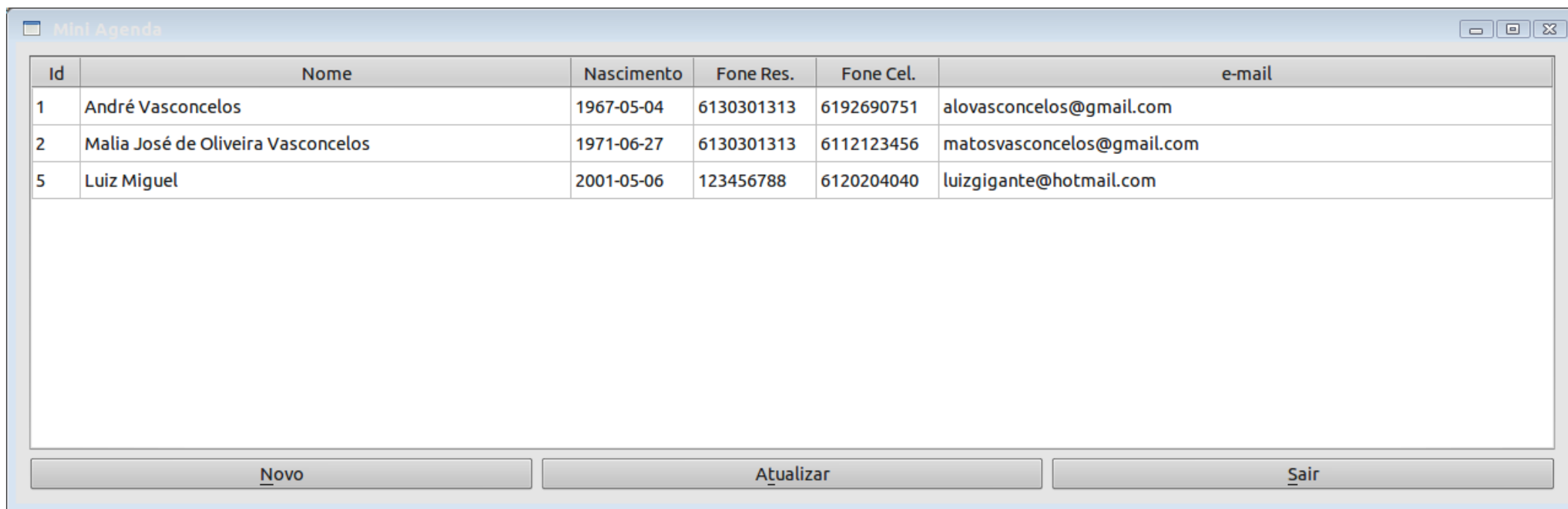


figura 47