Curso Linguagem C++

Referências

- Referências (só existem em C++), resolvem o problema do duplo acesso, sem que o programador precise preocupar-se com isso.
- Um ponteiro é uma nova memória (ocupando um novo endereço) e que visa a armazenar o endereço de uma outra memória.
- Uma referência é apenas um nome alternativo (um sinônimo) para um endereço de memória já existente.

Referências

- A criação desse sinônimo é conseguida através do operador de referência:
  - int a ;
  - int &ra = a; // 'ra' é uma referência (um sinônimo) para 'a'.
- Em outras palavras, se, por exemplo, a variável a for um apelido para o endereço 1000 da memória, então a variável ra será apenas um segundo apelido para esse mesmo endereço (1000).
- Isso significa que uma instrução que declara uma referência, como
  - int &ra = a ;
  - não cria uma nova memória e sim apenas um novo nome para uma memória já existente.

Referências

- Podemos então ter funções com parâmetros referência, ao invés de ponteiros. Nada será preciso fazer para minimizar duplos acessos. Para o programador, esse parâmetro será simplesmente um sinônimo para o argumento que será passado na chamada da função.

Inicialização de referências

- Regra básica:
  - Variáveis-Referência devem ser inicializadas, pois só podemos referenciar algo já existente. Assim, a referência já deve nascer associada àquela memória que ela pretende referenciar.
- Exemplos:
  - int a ;
  - int &ra = a ; //Correto! 'ra' foi inicializada.
  - int &rb ; //Errado! 'rb' não foi inicializada. Erro de compilação.

Diferenciando referências de ponteiros

- Embora possa parecer confuso que o operador &(referência) tenha o mesmo símbolo do operador &("address of"), na prática não há confusão pois eles sempre aparecem em posições diferentes e assim o compilador tem como analisá-los corretamente.

Diferenciando referências de ponteiros

- Quando se trata de &("adress of") o operador aparece sempre na posição à direita, de captura (leitura) de um endereço:
- Endereço:
  - int a = 5 ;
  - int * pa = &a ; // Aqui, o operador & captura o endereço de a (leitura).
- // Em seguida, o operador de atribuição [=]
- // armazena esse endereço em pa (gravação);

Diferenciando referências de ponteiros

- Quando se trata de &("referência") o operador aparece sempre na posição à esquerda, de recepção ou associação:
- Referência:
  - int a = 5 ;
  - int &ra = a ; // Aqui o operador & faz com que ra referencie a
  - // (tornando-se um um sinônimo). Pois o símbolo &
  - // não está na posição de leitura, e sim na posição de
  - // recepção ou associação, à esquerda.

Exemplo com referências

```
#include <iostream>
#include <limits>
bool validar_entrada( int & param_ref ) // 'param_ref' é uma referência
{
    std::cout << "\n-validar_entrada: testando conteúdo\n" <<
    endereço do parâmetro referencia 'param_ref'\n";
    std::cout << "conteúdo inicial de 'param_ref' = " << param_ref << '\n';
    std::cout << "endereço de 'param_ref' = " << &param_ref << '\n';
    std::cout << "\n-validar_entrada: entre com um numero inteiro: ";
    std::cin >> param_ref ; // alterando a memória referenciada
    if ( std::cin.fail() )
    {
        std::cout << "entrada invalida\n";
        std::cin.clear();
        std::cin.ignore( std::numeric_limits<int>::max() , '\n' );
        return false ; // entrada inválida
    }
    return true ; // entrada válida
}
```

"e

Exemplo com referências

```
int main()
{
    int x = 5 ; // 'x' é o apelido de um endereço de memória
    int * p = &x ; // 'p' é o apelido de um outro endereço de memória
    int & r = x ; // 'r' é uma referência para 'x': ou seja, um novo apelido
    std::cout << "conteúdo de 'x' = " << x << '\n';
    std::cout << "endereço de 'x' = " << &x << '\n';
    std::cout << "conteúdo de 'r' = " << r << '\n';
    std::cout << "endereço de 'r' = " << &r << '\n';
    std::cout << "conteúdo de 'p' = " << p << '\n';
    std::cout << "endereço de 'p' = " << &p << '\n';
    std::cout << "conteúdo APONTADO por 'p' = " << *p << '\n';'
    if ( validar_entrada ( x ) )
    std::cout << "\n-main: conteúdo de 'x' apos 'validar_entrada' =  << x << '\n';
    return 0;
}
```

Exemplo com referências

```
/* RESULTADO:
    Obs.: os endereços exibidos poderão variar,
    dependendo de plataforma e compilador. Mas a situação é a mesma.
    conteúdo de 'x' = 5
    endereço de 'x' = 0x22ff58
    conteúdo de 'r' = 5
    endereço de 'r' = 0x22ff58
    conteúdo de 'p' = 0x22ff58
    endereço de 'p' = 0x22ff54
    conteúdo APONTADO por 'p' = 5
    -validar_entrada: testando conteúdo
    e endereço do parâmetro referencia 'param_ref'
    conteúdo inicial de 'param_ref' = 5
    endereço de 'param_ref' = 0x22ff58
    -validar_entrada: entre com um numero inteiro: 20
    -main: conteúdo de 'x' apos 'validar_entrada' = 20 */
```

Exercício

- 1º: Criar um programa onde o usuário vai digitar 3 números e em seguida deve-se chamar uma **ÚNICA** função que vai dobrar esse valor e depois mostrará esses valores "dobrados". Esse primeiro exercício deve ser feito usando ponteiros.

TEMPO 30 MINUTOS

Exercício

- 2º: Criar um programa onde o usuário vai digitar 3 números e em seguida deve-se chamar uma **ÚNICA** função que vai dobrar esse valor e depois mostrará esses valores "dobrados". Esse segundo exercício deve ser feito usando referencia.

TEMPO 10 MINUTOS

Questões para revisão do Capítulo 6

- 1. Assinale as afirmações corretas, considerando o código abaixo:

  int x = 5;

  - int * px = &x;

- a. 'px' é uma referência para 'x'.
- b. 'px' é um ponteiro para 'x'.
- c. 'px' guarda o endereço de 'x'.
- d. O valor armazenado em 'px' é 5.
- e. O valor armazenado em 'x' é 5.
- f. É correto fazer: *x = 10;
- g. É correto fazer: *px = 10;

- 2. Assinale as afirmações corretas, considerando o código abaixo:

  int x = 5 ;

  int & rx = x ;

  - a. 'rx' é uma referência para 'x'.
  - b. 'rx' é um ponteiro para 'x'.
  - c. 'rx' guarda o endereço de 'x'.
  - d. O valor armazenado em 'rx' é 5.
  - e. O valor armazenado em 'x' é 5.
  - f. 'rx' é um sinônimo de 'x'.
  - g. É correto fazer: *rx = 10;
  - h. É correto fazer: x = 10;
  - i. É correto fazer: rx = 10;

- 3. Assinale as afirmações corretas, considerando o código abaixo:

  ```
  int x = 5 ;
  int * px ;
  px = &x ;
  ```

  - a. 'px' é um ponteiro para 'x'.
  - b. O valor apontado por 'px', a partir da terceira linha, é 5.
  - c. O código acima está incorreto e ocorrerá um erro de compilação.

- 4. Assinale as afirmações corretas, considerando o código abaixo:

```
int x = 5 ;
int & rx ;
rx = x ;
```

  - a. 'rx' é uma referência para 'x'.
  - b. O valor armazenado em 'rx', a partir da terceira linha, é 5.
  - c. O código acima está incorreto e ocorrerá um erro de compilação.

Capítulo 7
Iniciando orientação a objetos

Iniciando orientação a objetos

Orientação a objetos é uma das técnicas de programação suportadas por C++. Em C, é possível implementar essa técnica, mas ao custo de um trabalho extra considerável, já que ela não é suportada diretamente (nativamente) pela linguagem.

Iniciando orientação a objetos

Resumidamente, essa técnica é baseada em três princípios:

1º: Encapsulamento

- Recursos para proteger os dados com o código, de maneira inviolável.

2º: Herança

- Recursos para reaproveitamento de código.

3º: Polimorfismo

- Aqui temos também recursos para reaproveitamento de código, em um nível mais genérico.

Estruturas

- Quase sempre temos dados que se complementam (podendo até ser interdependentes) e que, assim, formam um conjunto de dados. Precisamos de uma maneira de expressar esse fato real no código.
- **Por exemplo:**
  - Uma data, é composta (pelo menos) de 3 informações:
  - dia
  - mes
  - ano

Estruturas

- Essas informações são **interdependentes**. Dependendo do **mês**, o último dia é 30 ou 31. E, para o mês de fevereiro, dependendo do **ano** o ultimo dia será 28 ou 29.

- Além disso, mesmo que não houvesse essa interdependência, sabemos que esses dados atuam em **conjunto** (complementares) para estabelecer uma **data completa**.

- E existem **regras**: regras para dia, regras para mês (entre 1 e 12) e regras que podem ser estabelecidas para o ano, dependendo da finalidade de uma data.

Estruturas

- Em **C**, poderíamos organizar esse conjunto de dados da seguinte forma:

```
struct Data
{
   int dia ;
   int mes ;
   int ano ;
} ;
```

Estruturas

- Uma estrutura (**struct**) permite agrupar informações que formam um conjunto. Ela é apenas um **modelo** para um **novo tipo de dados.** Esse tipo é constituído de **campos ou membros** (cada um dos dados agrupados). Esse novo tipo, pode ser usado para criar variáveis, as quais serão alocadas na memória de acordo com o modelo declarado pela **struct.**

Estruturas

- **Por exemplo:**

  // Declarando variáveis de diversos **tipos**:

  long x ;

  double d ;

  struct Data data_pagamento ;

  struct Data data_vencimento ;

- **Detalhando:**

Tipo	Variável	Comentário
long	**x**	'**x**' é uma variável do tipo **long**
double	**d**	'**d**' é uma variável do tipo **double**
struct Data	**data_pagamento**	'**data_pagamento**' é uma variável do tipo **struct Data**
struct Data	**data_vencimento**	'**data_vencimento**' é uma variável do tipo **struct Data**

Estruturas

- Após as declarações acima, podemos ilustrar esquematicamente a situação da **memória** com a seguinte tabela:

<table>
<tr><td>endereço (hipótese)</td><td>1000</td><td>1004
(1000+4)</td><td colspan="3">1012
(1004+8)</td><td colspan="3">1024
(1012+12)</td><td rowspan="3">1036
(1024 + 12)</td></tr>
<tr><td>mnemônico</td><td>x</td><td>d</td><td colspan="3">data_pagamento</td><td colspan="3">data_vencimento</td></tr>
<tr><td>composição</td><td>um único valor
(4 bytes)</td><td>um único valor
(8 bytes)</td><td>dia</td><td>mes</td><td>ano</td><td>dia</td><td>mes</td><td>ano</td></tr>
<tr><td colspan="3"></td><td colspan="3">(4+4+4 = 12 bytes em SO de 32 bits)</td><td colspan="3">(4+4+4 = 12 bytes em SO de 32 bits)</td></tr>
</table>

Estruturas

- **Atribuindo valores a essas variáveis:**

  x = 9 ; // '**x**' é um **único valor** (apenas um campo de informação)

  d = 8.7 ; // idem para '**d**'

  - Mas, ao contrário de '**x**' e '**d**', '**data_pagamento**' e '**data_vencimento**' não são constituídas por um único valor, sendo, ao contrário **um conjunto com 3 campos** de informação.
  - Então deve haver um meio de **acessar cada campo**. Isso é feito com um **ponto**.

- **Acessando cada campo das variáveis estruturadas** (o **ponto** acessa o campo)**:**

  data_pagamento.dia = 10 ; // acessa o campo '**dia**' (**pagamento**)

  data_pagamento.mes = 2 ; // idem para '**mes**'

  data_pagamento.ano = 2010 ; // e para '**ano**'

  data_vencimento.dia = 5 ; // acessa o campo '**dia**' (**vencimento**)

  data_vencimento.mes = 3 ; // idem para '**mes**'

  data_vencimento.ano = 2011 ; // e para '**ano**'

Estruturas

- E agora teremos na memória:

variável	x	d	data_pagamento			data_vencimento		
			dia	mes	ano	dia	mes	ano
valor	9	8.7	10	2	2010	5	3	2011

Estruturas

- **Sintetizando:**
  - Uma **struct** funciona como um descritor de campos. A estrutura é assim um **conjunto de campos**.
  - É algo muito **semelhante** a criação de uma tabela em um **banco de dados**.
  - Quando criamos uma nova tabela em um banco de dados, o que fazemos é justamente definir os **nomes e tipos** dos diferentes **campos**.
  - Para a linguagem, uma **struct** cria um novo **tipo de dados**. Assim, após declarar uma **struct** poderemos criar variáveis desse **tipo**.
  - E para acessar os dados internos da **struct** (isto é, seus **campos** ou **membros**):
    - usamos: o **operador de ligação de membro,** cujo símbolo é um **ponto**:
    - **data_pagamento.dia = 10 ;**
    - // a ligação do membro (ou campo) é simbolizada por um **ponto**

Estruturas e funções relacionadas

- É possível perceber, no exemplo acima, a utilidade da **struct.** O simples fato de agrupar os campos, permite expressar no código uma situação real: esses dados devem andar sempre juntos, em um único conjunto. Mas isso **ainda não é suficiente**.

- **Após** definir conjuntos de dados (**struct**), passamos a precisar de **funções especializadas no tratamentos desses dados**.

- Particularmente, é preciso garantir que sejam aplicadas as **regras** que devem regular os valores possíveis para cada campo.

Estruturas e funções relacionadas

- Precisamos, **no mínimo**, de funções que **validem** os campos. Além disso outras funções quase sempre são necessárias para atuar sobre cada conjunto de dados: imprimir, realizar cálculos, etc.

- No caso da **struct data**, além de **funções que garantam valores válidos** para dia, mês e ano, precisamos também de **outras funções** que implementem o tratamento de todas as necessidades relacionadas a uma data.

- Como, por exemplo, avaliar se uma data é maior que outra; ou calcular a diferença entre duas datas em número de dias, etc.

Limitação da linguagem C

- Na linguagem **C**, também escrevemos funções para atuar sobre cada **struct**. Contudo, em **C,** não há uma maneira de relacionar intimamente os dados com as funções. Para a linguagem essas são funções como outras quaisquer. Do ponto de vista do programador determinadas funções foram feitas para trabalhar com determinada **struct.** Mas, em **C**, não há como expressar essa necessidade real no código.

Limitação da linguagem C

- A consequência disso é que, em **C**, **não** podemos **garantir** (exceto com um trabalho extra considerável) que os **dados só serão alterados em funções que conhecem suas regras e impedem sua violação.** Ou seja: **qualquer função** poderá alterar os campos de uma **struct**.

Limitação da linguagem C

- **Exemplo:**

Poderíamos implementar a seguinte função, que recebendo uma data como argumento, altera os seus campos. Mas **analisa** se os dados enviados para alteração estão em conformidade com as regras.

```
struct Data
{
    int dia ;
    int mes ;
    int ano ;
    int ok ; // flag que indica se a data esta correta ou não,
} ;
struct Data alterar( struct Data dt , int dia , int mes , int ano )
{
    dt.dia = dia ;
    dt.mes = mes ;
    dt.ano = ano ;
    if ( < avaliação > ) // avalia se dia, mes e ano estão em // conformidade com as regras.
    {
        dt.ok = 1 ; // os valores estão corretos
    }
    else
    {
        dt.ok = 0 ; // condição de erro
    }
    return dt ;
}
```

- E o ***flag*** '**ok**' poderá ser usado em qualquer outra função que faça a leitura dos dados de uma data:

```
void imprimir ( struct Data dt )
{
    if ( dt.ok )
    {
        // os valores estão corretos: imprime a data normalmente
    }
    else
    {
        // condição de erro: imprime mensagem de erro.
    }
}
```

- Além disso seria preciso garantir que a data **não será impressa sem que tenha sido anteriormente alterada**, de modo que o campo **ok** contenha o resultado de uma avaliação, e não "lixo" (um valor arbitrário qualquer).

Limitação da linguagem C

- Para isso deveria haver uma função **inicializadora**. A inicialização garante um estado estável para uma estrutura desde o seu nascimento. Por exemplo, estabelecendo a condição de erro [**dt.ok = 0 ;**]. Ou utilizando valores *default* (caso existam).

- Em suma: **inicialização** é um elemento **indispensável** quando falamos em **encapsulamento**.

Limitação da linguagem C

- Contudo, mesmo que escrevêssemos uma função "**data_inicia**":

```
struct Data iniciar ( struct Data dt )
{
    // ...
    dt.ok = 0 ; // condição de erro
    return dt ;
}
```

Limitação da linguagem C

- Não há como garantir que ela sempre será chamada **imediatamente após** a criação de uma variável do tipo s**truct Data**. Tudo dependerá do programador. Se o programador **nunca esquecer** de chamar as funções adequadas, tudo correrá bem. Mas isso não é uma base segura para escrever programas, principalmente à medida em que eles crescem, tornando mais difícil assegurar que, em todas as partes do código, tudo está sendo feito corretamente.

Este exemplo mostra essa instabilidade potencial:

```
int main()
{
    struct Data pagamento ;
    struct Data vencimento ;
    pagamento = iniciar ( pagamento ) ;
    imprimir (pagamento ) ; // imprimira mensagem de erro: OK
    pagamento = alterar ( pagamento , 1, 10, 2010 );
    imprimir ( pagamento ) ; // imprimira a data ou erro: OK
    imprimir ( vencimento ); //O que será impresso aqui?
    // além disso, podemos:
    // pagamento.dia = ... ;
    // atribuir livremente qualquer valor a um campo  sem
    //usar a função adequada.
    return 0 ;
}
```