Programação orientada a objeto usando C++

Encapsulamento

A programação Orientada a Objetos (POO) gira ao redor do conceito de um objeto.

Os **objetos** são criados usando-se uma definição de classe.

Classe é um formato, de acordo com o qual os objetos são criados; um trecho de software que inclui a especificação de dados e as funções que operam sobre esses dados, e possivelmente sobre aqueles que pertencem a outras instâncias de classe.

As funções definidas em uma classe são chamados métodos, funções-membro ou **membros de função**.

As variáveis usadas em uma classe são os **membros de dados**.

O **encapsulamento de dados** é a combinação de dados e das operações relacionadas.

Objeto é uma instância de uma classe, entidade criada usando uma definição de classe.

Um objeto é definido pelos dados e pelas operações relacionadas e, como pode haver muitos objetos usados no mesmo programa, eles se **comunicam trocando mensagens** que revelam um para o outro tão poucos detalhes sobre suas estruturas internas quantos forem necessários para uma comunicação adequada.

Vantagens da POO:

- Forte acoplamento dos dados e das operações que pode ser bem usado na modelagem de eventos do mundo real;
- Os objetos permitem que os erros sejam descobertos mais facilmente, porque as operações são localizadas dentro de seus objetos. Mais fácil de rastrear os efeitos colaterais;
- Princípio de ocultamento de informação – os objetos nos permitem esconder, de outros objetos, certos detalhes de suas operações, de modo que essas operações não possam ser desfavoravelmente afetadas por outros objetos.

Um **objeto** é como uma caixa preta cujo **comportamento** é **muito bem definido**, e o usamos porque **sabemos o que faz**, não porque temos uma compreensão de como faz. Um objeto revela somente o necessário para o usuário utilizá-lo. Ele tem uma **parte pública** que pode ser acessada por qualquer usuário quando este envia uma mensagem que casa com qualquer dos nomes de função-membro revelados pelo objeto. **O usuário conhece somente os nomes dessas operações e o comportamento esperado**.

No caso de objetos, temos uma **declaração de classe e uma instância de objeto**. Por exemplo, na seguinte declaração de classe, C é uma classe e objeto1, objeto2 e objeto3 são objetos:

```
class C {
public:
        C  (char *s = "",  int i = 0,  double d = 1) {
                strcpy(dadosMembro1,s);
                dadosMembro2 = i;
                dadosMembro3 = d;
        }
        void funcaoMembro1 () {
                cout    << dadosMembro1 << ' ' << dadosMembro2 << ' '   << dadosMembro3 << endl;
        }

        void funcaoMembro2 (int i, char *s = "desconhecido") {
                dadosMembro2 = i;
                cout << i << " recebido de " << s << endl;
        }
protected:
        char dadosMembro1[20];
        int dadosMembro2;
        double dadosMembro3;
```

```
};

C objeto1 ("objeto1", 100, 2000),  objeto2 ("objeto2"), objeto3;
```

A Passagem de Mensagem equivale a uma chamada de função em linguagens tradicionais. No entanto, para **acentuar o fato de que em LOO as funções membro são relativas aos objetos**, este novo termo é usado. Por exemplo: a chamada à função *funçãoMembro1()* pública no objeto1:

objeto1.funcaoMembro1();

é vista como a mensagem *funcaoMembro1()* enviada ao objeto1. Ao receber a mensagem, o objeto invoca sua função-membro e exibe todas as informações relevantes. As mensagens podem incluir parâmetros, de modo que:

objeto1.funcaoMembro2(123);

é a mensagem *funcaoMembro2()* com o parâmetro 123 recebida pelo objeto1.

As linhas que contém essas mensagens podem estar no programa principal, em uma função ou em uma função-membro de outro objeto. Assim, o recebedor da mensagem é identificável, mas não necessariamente o expedidor. Se o objeto1 recebe a mensagem *funcaoMembro1()*, não sabe onde ela é originada. Somente responde a ela exibindo a informação que a *funcaoMembro1()* encapsula. Da mesma forma com a *funcaoMembro2()*. Desta forma, o expedidor pode preferir enviar uma mensagem que também inclua sua identificação, como em:

objeto1.funcaoMembro2(123, "objeto1");

Herança

As LOO permitem criar uma hierarquia de classes de modo que os objetos não tenham que ser instâncias de uma classe simples.

Vantagem principal da herança:

Reutilização do código – permite o uso de bibliotecas de classes criadas por outras pessoas. Sem modificá-las, é possível derivar outras classes que atendam às suas necessidades particulares. Uma classe derivada pode herdar características de mais de uma classe base (herança múltipla).

Relação entre a classe-base e uma classe derivada:

- A **parte pública** da classe-base está disponível à classe derivada e a qualquer objeto dessa classe.
- Qualquer **membro público** da classe-base é automaticamente um membro da classe derivada.
- **Nenhum membro privado** da classe-base pode ser acessado pela classe derivada.
- Além de **public** e **private**, há um outro especificador de acesso a membros denominado **protected:**

- Os membros de uma classe sempre podem ser acessados por outros membros da mesma classe, independentemente de serem **public** ou **private**. Entretanto, objetos desta classe, definidos fora dela, podem acessar somente os membros **public** da classe. Os membros **protected** são semelhantes a membros **private**, exceto pelo fato de que são visíveis a todos os membros de uma classe derivada; não podem ser acessados por nenhum objeto declarado fora dessas classes.

Se nenhum construtor for especificado na classe derivada, o construtor da classe-base será usado.

Herança pública e privada:

A declaração **public** indica que os membros públicos da classe-base serão membros públicos da classe derivada e os membros **protected** da classe-base serão membros **protected** da classe derivada. Os membros públicos da classe-base podem ser acessados por um objeto da classe derivada.

A declaração **private** indica que tanto os membros públicos quanto os protegidos da classe-base serão membros privados da classe derivada. Estes membros são acessíveis aos membros da classe derivada, mas não aos seus objetos. Portanto, um objeto da classe derivada não terá acesso a nenhum membro da classe-base.

Reescrevendo funções membro da classe base:

Quando se usa uma classe pré-definida e pré-compilada como classe-base, nem sempre as funções-membro dessa classe atendem completamente ao que se precisa.

É possível criar funções-membro de uma classe derivada que tenham o mesmo nome de funções-membro da classe-base. Isto faz com que a sintaxe da chamada a elas, por meio de um objeto, seja a mesma, independentemente de tratar-se de um objeto da classe-base ou da classe derivada. A nova função pode chamar a função da classe-base por meio do operador de resolução de escopo.

Exemplo 1: Um programa que mostra acesso público, privado e protegido à classe-base:

```
#include <iostream>

class BASE {
protected: int secreto;
private:   int ultra-secreto;
public:    int publico;
};

class DERIV1 : public BASE {
public:
        int a = secreto;        //Ok
        int b = ultra-secreto;  //ERRO: não acessível
        int c = publico;        //Ok
};
class DERIV2 : private BASE {
public:
        int a = secreto;        //Ok
        int b = ultra-secreto;  //ERRO: não acessível
        int c = publico;        //Ok
};

int main() {
        int x;

        DERIV1 obj1;            //DERIV1 é public
        x = obj1.a;             //ERRO: não acessível
        x = obj1.b;             //ERRO: não acessível
        x = obj1.c;             //Ok

        DERIV2 obj2;            //DERIV2 é private
```

```
        x = obj2.a;          //ERRO: não acessível
        x = obj2.b;          //ERRO: não acessível
        x = obj2.c;          //ERRO: não acessível
}
```

Exemplo 2: Exemplo do uso de herança para modelar uma base de dados de contas bancárias:

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <string.h>
using namespace std;
const int TAM=80;

class conta  {
private:
        int Nconta;          //Número da conta
        float saldo;
public:
        void getdata() {
                cout << "\n Conta:"; cin >> Nconta;
                cout << "\n Saldo: "; cin >> saldo;
        }
        void putdata() {
                cout << "\n Conta:" << Nconta;
                cout << "\n Saldo:" << saldo;
        }
        float Saldo()  { return saldo; }
};

class contaSimples : public conta
{};

class contaEspecial : public conta {
private:
        float limite;
public:
        void getdata() {
                conta :: getdata();
                cout << "\n Limite: ";  cin >> limite;
        }
        void putdata() {
                conta :: putdata();
                cout << "\n Limite:"  << limite;
                cout << "\n Saldo total:" << (Saldo()+limite);
        }
};

class contaPoupanca : public conta {
```

```
private:
float taxa;
public:
        void getdata() {
                conta :: getdata();
                cout << "\n Taxa: ";  cin >> taxa;
        }
        void putdata() {
                conta :: putdata();
                cout << "\n Taxa:"  << taxa;
                cout << "\n Saldo total:" << (Saldo()*taxa);
        }
};

int main()      {
        contaSimples c1, c2;
        contaEspecial c3;
        contaPoupanca c4;

        cout << "\n * Digite os dados da conta simples 1. ";
        c1.getdata();
        cout << "\n * Digite os dados da conta simples 2. ";
        c2.getdata();
        cout << "\n * Digite os dados da conta especial. ";
        c3.getdata();
        cout << "\n * Digite os dados da conta poupanca. ";
        c4.getdata();

        cout << "\n\n * Conta Simples 1. "; c1.putdata();
        cout << "\n\n * Conta Simples 2. "; c2.putdata();
        cout << "\n\n * Conta Especial 3. "; c3.putdata();
        cout << "\n\n * Conta Poupanca  "; c4.putdata();
        return 0;
}
```

Exemplo 3: Um programa que mostra herança simples e múltipla

```
class BaseClass {
public:
        BaseClass() { }
        void f (char *s = "unknown") {
                cout << "Function f() in BaseClass called from " << s << endl;
                h();
        }
protected:
        void g (char *s = "unknown") {
                cout << "Function g() in BaseClass called from " << s << endl;
        }
private:
        void h () {
                cout << "Function h() in BaseClass\n";
        }
```

```
};

class Derived1Level1 : public virtual BaseClass {
public:
        void  f  (char *s = "unknown") {
                cout << "Function f() in Derived1Level1 called from " << s << endl;
                g ("Derived1Level1");
                h ("Derived1Level1");
        }
        void h  (char *s = "unknown") {
                cout << "Function h() in Derived1Level1 called from " << s << endl;
        }
};

class Derived2Level1 : public virtual BaseClass {
public:
        void f(char *s = "unknown") {
                cout << "Function f() in Derived2Level1 called from " << s << endl;
                g("Derived2Level1");
                // h(); // error: BaseClass::h() is not accessible
        }
};

class DerivedLevel2 : public Derived1Level1, public Derived2Level1 {
public:
        void f(char *s = "unknown") {
                cout << "Function f() in DerivedLevel2 called from " << s << endl;
                g("DerivedLevel2");
                Derived1Level1::h("DerivedLevel2");
                BaseClass::f("DerivedLevel2");
        }
};

int main() {
        BaseClass bc;
        Derived1Level1 d1l1;
        Derived2Level1 d2l1;
        DerivedLevel2 dl2;
        bc.f("main(1)");
        // bc.g(); // error: BaseClass::g() is not accessible
        // bc.h(); // error: BaseClass::h() is not accessible
        d1l1.f("main(2)");
        // d1l1.g(); // error: BaseClass::g() is not accessible
        d1l1.h("main(3)");
        d2l1.f("main(4)");
        // d2l1.g(); // error: BaseClass::g() is not accessible
        // d2l1.h(); // error: BaseClass::h() is not accessible
        dl2.f("main(5)");
        // dl2.g(); // error: BaseClass::g() is not accessible
        dl2.h();
        return 0;
}
```

Saída do Programa:

```
Function f() in BaseClass called from main(1)
Function h() in BaseClass
Function f() in Derived1Level1 called from main(2)
Function g() in BaseClass called from Derived1Level1
Function h() in Derived1Level1 called from Derived1Level1
Function h() in Derived1Level1 called from main(3)
```

```
Function f() in Derived2Level1 called from main(4)
Function g() in BaseClass called from Derived2Level1
Function f() in DerivedLevel2 called from main(5)
Function g() in BaseClass called from DerivedLevel2
Function h() in Derived1Level1 called from DerivedLevel2
Function f() in BaseClass called from DerivedLevel2
Function h() in BaseClass
Function h() in Derived1Level1 called from unknown
```

A classe BaseClass é denominada *classe base* ou *superclasse*, e as outras, *subclasses ou classes derivadas* porque derivam da superclasse, por poderem usar os membros de dados e as funções-membro especificados na classe base (*BaseClass*) como **protected** (protegido) ou **public** (público). **Elas herdam todos esses membros a partir de sua classe base, de modo que não tenham que repetir as mesmas definições.** No entanto, uma classe derivada pode sobrepor a definição de uma função-membro introduzindo sua própria definição. Assim, tanto a classe base como a classe derivada têm alguma medida de controle sobre suas funções-membro.

A classe base pode decidir quais funções-membro e quais membros de dados podem ser revelados para as classes derivadas, de modo que o **princípio de ocultamento de informação** se mantenha não somente com relação ao usuário da classe base, mas também das classes derivadas. A classe derivada também pode decidir quais partes das funções-membro e dos membros de dados públicos e protegidos deve reter e usar e quais modificar. Por exemplo, tanto **Derived1Level1** como **Derived2Level1** definem suas próprias versões de **f()**. No entanto, o acesso à função-membro com o mesmo nome em qualquer das classes mais altas na hierarquia ainda é possível, precedendo-se a função com o nome da classe e com o operador de escopo, como na chamada de **BaseClass::f()** a partir de **f()** em **DerivedLevel2**.

Uma classe derivada pode adicionar alguns de seus novos membros. Tal classe pode se tornar uma casse base para outras classes que podem ser derivadas a partir dela, de modo que a hierarquia de heranças possa ser deliberadamente estendida. Por exemplo, a classe **Derived1Level1** é derivada de **BaseClass**, mas, ao mesmo tempo, é a classe base para a **DerivedLevel2.**

A **herança pública** (***public***) significa que os membros públicos da classe base são também públicos na classe derivada, e que os membros protegidos também assim o são. Em caso de **herança protegida (*protected*)** tanto os membros públicos como os protegidos da classe se tornam protegidos na classe derivada. Para a **herança privada (*private*)**, tanto os membros públicos como os protegidos da classe base se tornam privados na classe derivada. **Em todos os tipos de herança, os membros privados da classe base são inacessíveis para quaisquer classes derivadas.**

Os membros protegidos da classe base são acessíveis somente para as classes derivadas, e não para as não-derivadas. Logo, **Derived1Level1 e Derived2Level1** podem chamar a função-membro g() protegida da **BaseClass**, mas uma chamada para esta função a partir de **main()** é interpretada como ilegal.

Uma classe derivada pode ser derivada a partir de mais de uma classe base. Por exemplo, **DerivedLevel2** está definida como uma classe derivada de **Derived1Level1 e também de Derived2Level1**. No entanto, **DerivedLevel2** também herda as mesmas funções-membro da **BaseClass** duas vezes, porque ambas as classes usadas na definição de **DerivedLevel2** são derivadas a partir de **BaseClass**. Isso é redundante no melhor caso, e, no pior, pode causar o erro de compilação ***"member is ambiguous BaseClass::g() and BaseClass::g()."***, destacando a ambiguidade. Para evitar isto, as definições das duas classes incluem o modificador **virtual**, que significa que **DerivedLevel2** contém somente uma cópia de cada função-membro de **BaseClass**.

Ponteiros

Um ponteiro é um endereço de memória. Seu valor indica **onde uma variável está armazenada**, NÃO o que está armazenado. Proporciona um modo de acesso a uma variável sem referenciá-la diretamente. São variáveis auxiliares que nos permitem acessar os valores de outras variáveis indiretamente. Assim como todas as variáveis, também possuem dois atributos: um conteúdo e uma localização. Esta localização pode ser armazenada em outra variável, o que torna então um ponteiro para um ponteiro.

Os ponteiros são usados para:

1- Manipular elementos de matrizes;
2- Receber argumentos em funções que necessitem modificar o argumento original;

3- Passar strings de uma função para outra; usá-lo no lugar de matrizes;
4- Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde um item deve conter referências a outro;
5- Alocar e desalocar memória do sistema.

Endereços de memória:

O endereço é a **referência** que o computador usa para localizar variáveis.

A memória do computador é dividida em bytes, numerados de 0 até o limite de memória de sua máquina, chamados de endereços de bytes. Toda variável ocupa uma certa localização na memória, e seu endereço é do primeiro byte ocupado por ela.

Operador de endereços &: Usado para se conhecer o endereço ocupado por uma variável.
Operador indireto (*): operador unário que resulta no valor da variável apontada.

Exemplo: uso do operador &

```
#include <iostream>
using namespace std;
int main() {
   int i, j, k;
   cout << "\n Endereco de i: " << &i;
   cout << "\n Endereco de j: " << &j;
   cout << "\n Endereco de k: " << &k;
}
```

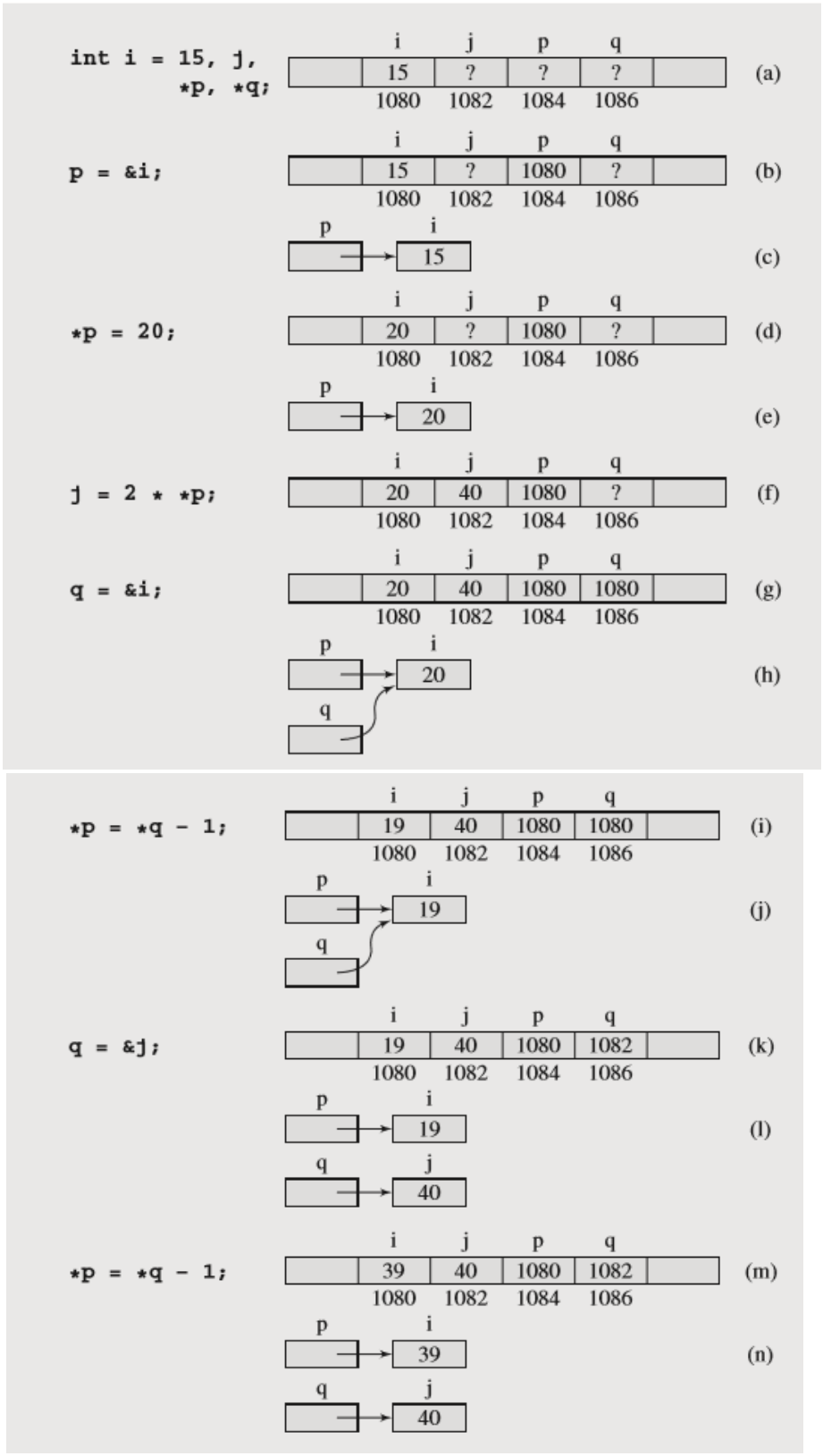
Saída:

```
Endereco de i:  0x6dfefc
Endereco de j:  0x6dfef8
Endereco de k: 0x6dfef4
```

O endereço ocupado por essas variáveis dependerá de vários fatores: tamanho do sistema operacional, se há ou não outros programas residentes na memória, etc. Logo, diferentes valores podem ser encontrados.

O operador **<<** imprime endereços em hexadecimal, precedidos pelo prefixo **0x**. Cada endereço difere do próximo por 4 bytes – já que inteiros geralmente ocupam 4 bytes de memória. Essa diferença depende do tipo da variável. Os endereços aparecem em **ordem descendente**, pois **variáveis automáticas são guardadas na pilha**.

Exemplos de instruções contendo variáveis ponteiro e de como os valores são armazenados na memória do computador (obs: neste exemplo o inteiro ocupa 2 bytes):



Uma variável aponta para outra variável quando a primeira contém o endereço da segunda.

Ponteiros e variáveis de referência

Considere as seguintes declarações: int n=5, *p = &n, &r=n;

A variável **p** é do tipo **int***, um ponteiro para um inteiro.
A variável **r** é do tipo **int&**, uma variável de referência inteira.

A variável de referência deve ser inicializada como uma referência a uma variável particular, e esta referência não pode ser mudada e não pode ser nula!

A variável de referência r pode ser considerada um **nome diferente para a variável n**. Logo, se **n** muda, **r** também muda. É implementada como um ponteiro constante à variável.

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

void  main() {
   int n = 5, *p = &n, &r = n;
   cout << n << ' ' << *p << ' ' << r << endl;  // 5 5 5
   n = 7;  cout << n << ' ' << *p << ' ' << r << endl;  // 7 7 7
   *p = 9; cout << n << ' ' << *p << ' ' << r << endl;  // 9 9 9
   r = 10; cout << n << ' ' << *p << ' ' << r << endl;  // 10 10 10
}
```

Passagem de argumentos por referência:

O operador de referência cria outro nome para uma variável já existente. Toda operação em qualquer dos nomes tem o mesmo resultado. Uma referência não é uma cópia da variável a que se refere. É a mesma variável sob diferentes nomes.

Vantagens: a função pode acessar as variáveis da função que chamou; uma função pode retornar mais de um valor para a função que chama. Os valores a serem retornados são colocados em referência de variáveis da função chamadora.

Nos argumentos passados por valor, a função chamada cria novas variáveis do mesmo tipo dos argumentos e copia nelas o valor dos argumentos passados. Assim, a função não tem acesso às variáveis originais da função que chamou, portanto, não as pode modificar.

Exemplo: solicitar ao usuário para inserir o preço atual de uma mercadoria, modificando o preço para um valor reajustado em 20% e calculando o valor do aumento.

```
void reajusta20(float &p, float &r) {
        r = p * 0.2;
        p *= 1.2;
}
void main() {
      float preco, valor_reajuste;
      do {
          cout << "\n\n Insira o preco atual:";
          cin >> preco;
          reajusta20(preco, valor_reajuste);
          cout << "\n preco novo:" << preco;
          cout << "\n Aumento:"    << valor_reajuste;
        } while (preco != 0.0);
}
```

O operador & é usado apenas na definição do tipo do argumento. A chamada a uma função que recebe uma referência é idêntica à chamada às funções em que os argumentos são passados por valor. A declaração: **float &p**,

float &r indica que **p** e **r** são outros nomes para as variáveis passadas como argumento pela função que chama. Ou seja, quando usamos **p** e **r**, estamos realmente usando preco e valor_reajuste de main().

Passando argumentos por referência com ponteiros:

A função chamadora, em vez de passar valores para a função chamada, passa endereços usando o operador de endereços. Estes endereços são de variáveis da função chamadora onde queremos que a função coloque novos valores.

A função chamada deve criar variáveis para armazenar os endereços enviados pela função chamadora. Estas variáveis são ponteiros.

```
void reajusta20(float *p, float *r) {
   *r = *p * 0.2;
   *p *= 1.2;
}

void main() {
   float preco, valor_reajuste;
   do {
      cout << "\n\n Insira o preco atual:";
      cin >> preco;
      reajusta20(&preco, &valor_reajuste);
      cout << "\n preco novo:" << preco;
      cout << "\n Aumento:"    << valor_reajuste;
   } while (preco != 0.0);
}
```

Um ponteiro para um tipo ***float*** não é do tipo ***float***, mas sim do tipo ***float****. O asterisco faz parte do nome do tipo e indica **ponteiro para**. A declaração dos argumentos da função reajusta20(float *p, float *r) indica que ***p** e ***r são** do tipo ***float*** e que ***p* e *r*** são ponteiros para variáveis ***float*.**

Usar *p é uma maneira indireta de usar a variável **preco** de **main()**. Toda alteração feita em ***p afetará** diretamente **preco**. Como **p** contém o endereço da variável **preco**, dizemos que **p** aponta para **preco.**

Ponteiros sem funções:

```
int main()  {
   int x=4, y=7;
   cout << "\n &x= " << &x << "\t x=" << x;
   cout << "\n &y= " << &y << "\t y=" << y;

   int *px, *py;        //Ponteiros para variáveis int
   px = &x;
   py = &y;
   cout << "\n px= " << px << "\t *px=" << *px;
   cout << "\n py= " << py << "\t *py=" << *py;
}
```

Saída:

```
&x= 0x6dfef4    x=4
```

```
&y= 0x6dfef0    y=7
px= 0x6dfef4    *px=4
py= 0x6dfef0    *py=7
```

Ponteiros e variáveis apontadas:

```
void main() {
   int x, y;
   int *p=&x;    //Inicializa p com o endereço de x

   *p = 14;      //O mesmo que x=14;
    y = *p;      //O mesmo que y=x; o operador indireto * indica o valor da variável apontada.

   cout << "\n y= " << y;
}
```

Saída: y = 14

Operações com ponteiros:

```
void main() {
   int x=5, y=6;
   int *px, *py;
   px = &x;          //Atribuições de um endereço a um ponteiro
   py = &y;

   cout << "\n px = " << px     //Nome do ponteiro: valor contido nele - endereco da variável apontada
      << ", *px = " << *px      //Op. Indireto
      << ", &px = " << &px;     //Op. Endereços

   cout << "\n py = " << py
      << ", *py = " << *py      //Op. Indireto - valor da variável apontada
      << ", &py = " << &py;    //Op. Endereços

   if (px < py)                 //Comparações - sempre entre ponteiros de mesmo tipo
            //Diferenca entre ponteiros-expressa em número de tipo apontado entre eles
     cout << "\n py-px= " << (py - px);
   else
     cout << "\n px-py= " << (px - py);

   py++;                        //Incremento de um int.  Movimenta para o proximo tipo apontado.

   cout << "\n\n py++ ";
   cout << "\n py = " << py
      << ",*py = " << *py      //Op. Indireto
      << ",&py = " << &py;    //Op. Endereços

   px = py + 3;                //Caminha três inteiros adiante de py.

   cout << "\n\n px = py+3 ";
   cout << "\n px = " << px
      << ", *px = " << *px           //Op. Indireto
      << ", &px = " << &px;         //Op. Endereços

   cout << "\n px-py= " << (px - py);
}
```

Saída:

```
px = 0x6dfeec, *px = 5, &px = 0x6dfee4
py = 0x6dfee8, *py = 6, &py = 0x6dfee0
px-py= 1

py++
py = 0x6dfeec,*py = 5,&py = 0x6dfee0

px = py+3
px = 0x6dfef8, *px = 7208852, &px = 0x6dfee4
px-py= 3
```

A unidade adotada em operações com ponteiros:

Quando declaramos um ponteiro, o compilador necessita conhecer o tipo da variável apontada para poder executar corretamente operações aritméticas.

int *pi; **double *pd;** **float *pf;**

O tipo declarado é entendido como o tipo da variável apontada. Assim, se somarmos 1 a **pi**, estaremos somando 4 bytes (um int). Se somarmos 1 a **pd** estaremos somando 8 bytes (um double), e assim por diante. A unidade com ponteiros é o número de bytes do tipo apontado.

Ponteiros e matrizes

O compilador transforma matrizes em ponteiros na compilação, pois a arquitetura do microcomputador compreende ponteiros, e não matrizes. Qualquer operação feita com índices de uma matriz pode ser feita com ponteiros.

O nome de uma matriz representa seu endereço de memória. Este endereço é o endereço do primeiro elemento da matriz. Ou seja, o nome de uma matriz é um ponteiro que aponta para o primeiro elemento da matriz.

```
void main() {

   int M[5]={92, 81, 70, 69, 58};

   cout << "\n Imprime os valores dos elementos de uma matriz.";
   for (int i=0; i<5; i++) {
         cout << "\n " << M[i];
   }

   cout << "\n Imprime os valores dos elementos de uma matriz usando notação ponteiro";
   for (int i=0; i<5; i++)
   {
         cout << "\n " << *(M+i);
   }
}
```

A expressão *(M+i) tem exatamente o mesmo valor de **M[i]**. **M** é um ponteiro para **int** e aponta para **M[0]**. Da aritmética com ponteiros, se somarmos **1 a M**, obteremos o endereço de **M[1]**, **M+2** é o endereço de **M[2]**, etc. Regra geral:

M + i é equivalente a **&M[i]**, portanto ***(M + i)** é equivalente a **M[i]**.

Ponteiros constantes e ponteiros variáveis:

Um ponteiro variável é um lugar na memória que armazena um endereço. Um ponteiro constante é um endereço, uma simples referência.

A expressão: cout << "\n " << ***(M+i)**; NÃO pode ser substituída por cout << "\n " << ***(M++)**; **pois não** podemos incrementar uma constante!

Assim como existem inteiros constantes e inteiros variáveis, existem ponteiros constantes e ponteiros variáveis. **O nome de uma matriz é um ponteiro constante e não pode ser alterado.** Isso vale para qualquer constante. Escrever x=3++; é errado!! O compilador apresentaria um erro.

```
void main() {
   int M[5]={92, 81, 70, 69, 58};
   int *p=M;  //Ponteiro para int inicializado com o nome da matriz
   for (int i=0; i<5; i++)  {
      cout << "\n " << *(p++);
   }
}
```

O ponteiro p contém inicialmente o endereço do primeiro elemento da matriz M[0]. Para acessar o próximo elemento, basta incrementar **p** de um. Após o incremento, **p** aponta para o próximo elemento, **M[1]**, e a expressão ***(p++)** representa o conteúdo do segundo elemento. Os elementos da matriz são acessados em ordem.

Passando matrizes como argumento para funções:

```
float media (float *lista, int tamanho) {
   float m=0;
   for (int i=0; i<tamanho; i++)
      m+= *lista++;
   return (m/tamanho);
}
void main() {
   const int MAXI=20;
   float notas[MAXI];
   int i;

   for (i=0; i<MAXI; i++) {
      cout << "Digite a nota do aluno: " << (i+1) << ": ";  cin >> *(notas+i);
      if ( *(notas+i) < 0 ) break;
   }
   float m = media (notas, i);
   cout << "\n\n Média das notas: " << m;
}
```

A instrução *lista++ é interpretada como *(lista++) e o ponteiro é incrementado.

Ponteiros e Strings:

Strings são matrizes do tipo **char.** Assim, a notação ponteiro pode ser aplicada a **strings** do mesmo modo que é aplicada a matrizes.

Exemplo de uma função que procura um caractere numa cadeia de caracteres. Esta função retorna o endereço da primeira ocorrência do caractere, se este existir, ou o endereço zero caso o caractere não seja encontrado.

```
char * procura (char *s, char ch);
void main() {
      setlocale(LC_ALL, "");
      char ch, str[81], *ptr, letra;

      cout << "\n Digite uma frase: ";          gets (str);
```

```
        cout << "\n Digite um caractere: ";        cin >> letra;

        ptr = procura (str, letra);
        cout << "\n A frase começa no endereço: " << unsigned(str);

        if (ptr)  {
           cout << "\n Primeira ocorrência do caractere " << letra << ": " <<  unsigned(ptr);
           cout << "\n A sua posição é: " << unsigned(ptr-str);
        }else
           cout << "\n O caractere " << letra << " não existe nesta frase";
}
char * procura (char *s, char ch) {
   while (*s != ch && *s != '\0')  s++;
   if (*s != '\0')   return s;
   return (char *)0;
}
```

O endereço da matriz **str** é usado como argumento na chamada à função **procura()**. Este endereço é uma constante, mas, quando passado por valor, uma variável é criada para armazenar uma cópia dele. A variável é um ponteiro **char** de nome **s.** A expressão **s++** avança o ponteiro para sucessivos endereços dos caracteres da cadeia.

Exemplo de saída:

Digite uma frase: Eu sou feliz!
Digite um caractere: u
A frase começa no endereço: 2686619
Primeira ocorrência do caractere u: 2686620
A sua posição é: 1

Funções de Biblioteca para manipulação de strings

Exemplo:

```
int strlen (char *s);
void strcpy (char *dest, char *orig);
int strcmp (char *s, char *t);

//Retorna o tamanho da cadeia
int strlen (char *s) {
   int i=0;
   while (*s)  {i++; s++;}
   return i;
}

//Copia a cadeia origem na cadeia destino
void strcpy (char *dest, char *orig) {
   while (*dest++ = *orig++);
}

//Compara a cadeia s com a cadeia t.   Retorna a diferença ASCII:
//  um número positivo se s>t,  um número negativo se s<t e zero se s==t
int strcmp (char *s, char *t) {
   while (*s==*t && *s && *t)   {s++; t++;}
   return *s - *t;
}
void main() {
   char s1[]="Oi querida";
```

```
   char s2[]="O que foi?";
   cout << strlen("João e Maria");
   cout << "\ns1==s2?" << strcmp(s1,s2);
   strcpy(s1, s2);
   cout << "\n" << s1;
   cout << "\ns1==s2?" << strcmp(s1,s2);
}
```

Ponteiros para uma cadeia de caracteres constante:

```
//Mostra duas diferentes inicializações de strings
void main() {
   setlocale(LC_ALL,"");
   char s1[] = "Saudações!";      //Ponteiro constante: não pode ser alterado
   char *s2  = "Saudações!";      //Ponteiro variável

   cout << "\n" << s1;
   cout << "\n" << s2;

   //s1++;                  //Erro! Não podemos incrementar uma constante
   s2++;                    //OK! Aponta para o próximo caractere
   cout << "\n" << s2;     //Imprime audações
}
```

Alocação e desalocação de memória: new e delete

Operador new: para aquisição de memória em tempo de execução. Obtém memória do sistema operacional e retorna um ponteiro para o primeiro byte do novo bloco de memória que foi alocado. Uma vez alocada, esta memória continua ocupada até que seja desalocada explicitamente pelo operador **delete.**

```
class String  {
private:
   char *str;
public:
   String() {
      str = new char;     //Reserva um único byte na memória.
      *str='\0';
   }
   String (char *s) {
      /*Retorna um ponteiro para um bloco de memória do tamanho exato para armazenar
         a cadeia s mais o '\0'. */
      str = new char[strlen(s)+1];
      strcpy(str,s);
   }
   ~String() {
      if (str)    delete str;
   }
   void print()   { cout << str; }
};

void main() {
   setlocale(LC_ALL,"");
   String s="A vida é para ser vivida";
   String s1;
   cout << "\ns="; s.print();
   cout << "\ns1="; s1.print();
}
```

Se não houver memória suficiente para satisfazer a exigência da instrução, **new** devolverá um ponteiro com o valor **zero** (**NULL**). Antes de liberar a memória alocada por **new**, devemos verificar se essa memória foi realmente alocada. Se **str** aponta para o endereço **zero,** nada é liberado.

Liberar a memória não libera o ponteiro que aponta para ela. Não é mudado o endereço contido no ponteiro; simplesmente esse endereço não será mais válido. Assim, não se deve usar um ponteiro para uma memória que foi liberada.

Ponteiros void:

Quando atribuímos um endereço a um ponteiro, este deve ser do mesmo tipo do ponteiro. Por exemplo, não devemos atribuir um endereço de uma variável **int** a um ponteiro **float**. Mas há uma exceção.

Os ponteiros do tipo void: ponteiro de propósito geral que pode apontar para qualquer tipo de dado..

Declaração **void *p;** //p aponta para qualquer tipo de dado

Usados em situações em que seja necessário que uma função retorne um ponteiro genérico e opere independentemente do tipo de dado apontado.

Obs: o conceito de ponteiros **void** não tem nada a ver com o tipo **void** para funções.

Qualquer endereço pode ser atribuído a um ponteiro void. Exemplo:

```
void main() {
        int i=5, *pi;
        void *pv;               //Ponteiro genérico
        pv = &i;                //Endereço de um int

        //cout << *pv;          //ERRO: void is not a pointer-to-object type
        pi = (int *)pv;

        cout << *pi;    //OK
}
```

O conteúdo da variável apontada por um ponteiro **void** não pode ser acessado por meio deste ponteiro. Deve-se criar outro ponteiro e fazer a conversão de tipo na atribuição.

Redimensionando strings

É possível escrever uma função membro para a classe **string** que muda o tamanho da cadeia de caracteres. Exemplo com uma função operadora +=.

```
class String {
private:
  char *str;
public:
  String() {
    str = new char;
    *str='\0';
  }
  String (char *s) {
    str = new char [strlen(s)+1];
    strcpy (str, s);
  }
  void operator += (const char *s)
  {
    char *temp;
    int tamanho = strlen (str)+strlen(s);
    temp = new char [tamanho+1];
```

```
        strcpy (temp, str);
        strcat (temp,s);
        delete str;
        str = temp;
    }
    ~String()
    {
        if (str)    delete str;
    }
    void print() {cout << str;}
};

void main() {
    String s1("Feliz Aniversário! ");
    s1+="Joana";
    cout << "\n";   s1.print();
}
```

A função operator += acrescenta o conteúdo de uma cadeia de caracteres, recebida como argumento, a um objeto **string** já existente. O objeto **string s1** é redimensionado dinamicamente. Todos os detalhes do redimensionamento estão embutidos na função-membro.

Dimensionando matrizes em tempo de execução

```
float media (float *lista, int tamanho);
void main() {
        int tamanho;
        float *notas;
        cout << "\n Qual o número de notas? ";       cin >> tamanho;
        notas = new float (tamanho);
        for (int i=0; i<tamanho; i++) {
                cout << "Digite a nota do aluno " << (i+1) << ":";
                cin >> *(notas+i);
        }
        float m = media (notas, tamanho);
        cout << "\n\nMédia das notas: " << m;
        delete [] notas;
}

float media (float *lista, int tamanho) {
        float m=0;
        for (int i=0; i<tamanho; i++)
                m+=*(lista+i);   // ou m+=*lista++;
        return m/tamanho;
}
```

Ponteiros para objetos

Os ponteiros podem apontar para objetos da mesma forma que apontariam para qualquer tipo de dado.

Muitas vezes quando se está escrevendo um programa, se desconhece o número de objetos a serem criados. Neste caso, usa-se **new** para criar objetos em tempo de execução. O operador **new** retorna um ponteiro para um objeto sem nome.

```
class Venda {
private:
   int npecas; float preco;
public:
```

```
    void getvenda() {
      cout << "\n Insira o no.de pecas:";  cin>> npecas;
      cout << "\n Insira o preco:";  cin>> preco;
    }
    void printvenda() const {
      cout << "\n No. de pecas:"  << npecas;
      cout << "\n Preco:" << preco;
    }
};

int main ()  {
    Venda A;
    A.getvenda();
    A.printvenda();

    Venda *B;
    B = new Venda;
    B -> getvenda();
    B -> printvenda();
}
```

A função **main()** cria dois objetos da classe **Venda**: o objeto A e um objeto sem nome, apontado pelo ponteiro B. Nesse caso, a forma de acesso aos membros de um objeto é por meio do seu endereço, e não de seu nome. O operador de acesso a membros (->) conecta um ponteiro para um objeto a um membro dele, enquanto que o operador ponto (.) conecta o nome de um objeto a um membro dele.

Usando referências

Criar uma referência a um objeto definido pelo operador **new**.

```
class Venda {
private:
    int npecas; float preco;
public:
    void getvenda() {
      cout << "\n Insira o no.de pecas:";  cin>> npecas;
      cout << "\n Insira o preco:";  cin>> preco;
    }
    void printvenda() const {
      cout << "\n No. de pecas:"  << npecas;
      cout << "\n Preco:" << preco;
    }
};

int main () {
    Venda& A = *(new Venda);
    A.getvenda();
    A.printvenda();
}
```

A expressão **new Venda** retorna um ponteiro para uma área de memória, grande o suficiente para armazenar um objeto **Venda**. O objeto original pode ser referenciado por meio da expressão:

***(new Venda);**

Este é o objeto apontado pelo ponteiro. Criamos uma referência a este objeto de nome **A. Assim, A** é o próprio nome do objeto e seus membros podem ser acessados usando o operador ponto em vez do operador **->.**

Uma matriz de ponteiros para objeto:

Geralmente, quando um programa necessita manipular um grupo de objetos, é mais flexível a criação de uma matriz de ponteiros para objetos do que uma matriz para agrupar os próprios objetos. Por exemplo, é mais rápido indexar uma matriz de ponteiros para objetos do que indexar os próprios objetos.

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

using namespace std;

class Nome  {
private:
   char *str;
public:
   int getnome() {
      char nome[100];
      gets(nome);
      str = new char [strlen(nome)+1];
      strcpy (str, nome);
      return strcmp (str, "");
   }
   void print() { cout << str; }
};
int main() {
   Nome *p[80];
   int n, i;
   for (n=0; ;  n++)  {
      cout << "\nDigite nome ou [ENTER] para fim: ";
      p[n] = new Nome;
      if (p[n] -> getnome()==0)   break;     //equivalente a (*(p+n))-> getnome()
   }
   cout << "\n\n Lista dos nomes:";
   for (i=0; i<n; i++)
      cout << "\n"; p[i]->print();
}
```

Ponteiros para ponteiros

Criar uma matriz de ponteiros para objetos e mostrar como ordenar esses ponteiros baseados nos dados contidos nos objetos. Flexibilidade na criação e ordenação de classes complexas.

```
enum Boolean {False, True};

class String {
private:
   char *str;
public:
   int getname() {
      char nome[100];
      gets(nome);
      str = new char [strlen(nome)+1];
      strcpy (str, nome);
      return strcmp (str,"");
   }
   Boolean operator > (String s)
   {   return (strcmp(str, s.str)>0)? True:False; }
```

```
    void print() { cout << str; }
};

void ordena(String **p, int n);

int main() {
   //Cria uma matriz de 100 ponteiros para objetos do tipo String
   String *p[100];
   int n, i;

   for (n=0; ;  n++)  {
      cout << "\nDigite nome ou [ENTER] para fim: ";
      //Cria um objeto por vez com o operador new
      //Toda vez que um objeto é criado, seu endereço é atribuído a um elemento da matriz de ponteiros
      p[n] = new String;
      //Processo interrompido quando usuário digita uma string vazia
      if (p[n] -> getname()==0)   break;
   }
   cout << "\n\n Lista original:";
   for (i=0; i<n; i++) {
       cout << "\n "; p[i]->print();
   }

   ordena (p,n);

   cout << "\n\n Lista Ordenada:";
   for (i=0; i<n; i++)  {
      cout << "\n"; p[i]-> print();
   }
}

//Ordena ponteiros para os objetos
void ordena(String **p, int n)
{
   String *temp;
   for (int i=0; i<n; i++)
   {
      for (int j=i+1; j<n ; j++)
      {
         if (*(*(p+i)) > *(*(p+j)))   //ou: if (*p[i] > *p[j])
         {
            temp = *(p+i);
            *(p+i) = *(p+j);
            *(p+j) = temp;
         }
      }
   }
}
```

Ordenando ponteiros. A Função ordena não ordena objetos, e sim os ponteiros para os objetos. Esta ordenação é muito mais rápida que a ordenação dos próprios objetos, pois estaremos movimentando ponteiros pela memória, e não objetos. Os ponteiros são variáveis que ocupam pouco lugar de memória, entretanto os objetos podem ocupar muita memória.

Em **String **p;** o tipo da variável **p** é **String****.** Indica que **p** é um ponteiro duplamente indireto. Quando o endereço de um objeto da classe **String** é passado para uma função como argumento, o seu tipo é **String***. Um único asterisco é usado para indicar o endereço de um objeto.

A função **ordena** não recebe o endereço de um objeto, e sim o endereço de uma matriz de ponteiros para objetos do tipo **String**. O nome de uma matriz é um ponteiro para um elemento dela. Neste caso, o nome da matriz é um ponteiro que aponta para outro ponteiro.

Saída do programa:

```
Digite nome ou [ENTER] para fim: Maria Joao
Digite nome ou [ENTER] para fim: Jose Antonio
Digite nome ou [ENTER] para fim: Ana Carolina
Digite nome ou [ENTER] para fim:

Lista original:
Maria Joao
Jose Antonio
Ana Carolina

Lista Ordenada:
Ana Carolina
Jose Antonio
Maria Joao
```

Ponteiros para funções

Tipo de ponteiro especial: ponteiro que aponta para uma função.

Se você definiu um ponteiro para função e atribuiu a ele o endereço de uma função particular, dizemos que este ponteiro aponta para a função e que a função pode ser executada por meio do ponteiro.

```
void doisbeep(void)   {          //Toca o autofalante duas vezes
   cout << '\x07';
   for (int i=0; i<5000;  i++)
      cout << '\x07';
}
int main()   {
   void (*pf) (void);        //pf: ponteiro para uma função void
   pf = doisbeep;                    //Sem parênteses - atribui o endereço da função
   (*pf)();                  //Chama a função - equivalente a doisbeep();
}
```

O nome de uma função, desacompanhado de parênteses, é o seu endereço.

Um dos atributos de uma variável – seu endereço indicando sua posição na memória do computador.

Um dos atributos de uma função – o endereço indicando a localização do corpo da função na memória.

Após a chamada a função, o sistema transfere controle a essa localização para executar a função. Logo, é possível usar ponteiros para funções! Úteis para 'funcionais' – funções que tomam funções como argumentos, ex. integrais.

Exemplo:

```
double f (double x)  {
        return 2*x;
}
```

f: ponteiro para a função f()

***f**: a própria função

(*f)(7): uma chamada à função

Exemplo para calcular uma soma: $\sum_{i=n}^{m} \boldsymbol{f}(\boldsymbol{i})$

Para fazer a soma, devemos fornecer, além dos limites **n** e **m**, uma função **f**. Solução:

```
double soma (double (*f)(double), int n, int m)   {
        double resultado = 0;
        for (int i=n; i<=m; i++)  {
                resultado += f(i);
```

```
    return resultado;
}
```

A declaração double (*f)(double), significa que **f** é um ponteiro para uma função com argumento *double* e valor de retorno *double*.

Exemplo de chamada:

```
cout << sum (f,1,5)    << endl;
cout << sum (sin, 3,5) << endl;
```

Lista de Exercícios:

1) O que significa o operador asterisco em cada um dos seguintes casos:

a) int *p;
b) cout << *p;
c) *p=x*5;
d) cout << *(p+1).

2) Qual o resultado do programa abaixo:

```
#include <iostream>
using namespace std;
int main()  {
    float i=3, j=5;
    float *p=&i, *q=&j;
    cout << p==q;
    cout << "\n"  << *p - *q;
    cout << "\n"  << **&p;
    cout << "\n"  << 3*-*p / *q +7;
}
```

3) Qual a saída deste programa?

```
#include <iostream>
void main() {
    int i=5, *p;
    p=&i;
    cout << p << '\t' << (*p+2) << '\t' << **&p << '\t' << (3**p) << '\t' << (**&p+4);
}
```

4) Se *i* e *j* são variáveis inteiras e *p* e *q*, ponteiros para *int*, quais das seguintes expressões de atribuição são incorretas?

a) p = &i;
b) *q = &j;
c) p = &*&i;
d) i = (*&)j;
e) i = *&*&j;
f) q=&p;
g) i = (*p)++ + *q;
h) if (p==i) i++;

5) Admitindo a declaração: int mat[8]; por que a instrução mat++; é incorreta?

6) Admitindo a declaração: int mat[8]; quais das seguintes expressões referenciam o valor do terceiro elemento da matriz?

a) *(mat+2); b) *(mat+3); c) mat+2 d) mat+3

7) O que faz o programa seguinte?

```
#include <iostream>
void main() {
        int mat[] = {4,5,6};
        for (int j=0; j<3; j++)
                cout << "\n" << *(mat+j);
}
```

8) O que faz o programa seguinte?

```
#include <iostream>
void main() {
        int mat[] = {4,5,6};
        for (int j=0; j < 3;  j++)
                cout << "\n" << (mat+j);
}
```

9) O que faz o programa abaixo?

```
#include <iostream>
void main() {
        int mat[] = {4,5,6};
        int *p=mat;
        for (int j=0;  j < 3;  j++)
                cout << "\n" << *p++;
}
```

10) Assumindo a declaração: ***char *s="Eu vou estudar";***

O que imprimirão as instruções seguintes:

a) cout << s; b) cout << &s[0]; c) cout << (s+11); d) cout << s[0];

11) O operador new:

a) cria uma variável de nome new;
b) retorna um ponteiro void;
c) Aloca memória para uma nova variável;
d) Informa a quantidade de memória livre.

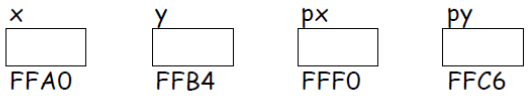
12) O operador delete:

a) apara um programa;
b) devolve memória ao sistema operacional;
c) diminui o tamanho do programa;
d) Cria métodos de otimização.

13) Assumindo que o endereço da variável x foi atribuído a um ponteiro px, escreva uma expressão que não usa x e que divida x por 5.

14) Dadas as declarações abaixo, qual é o valor dos itens?

```
int  x = 10,  *px = &x;
float  y = 5.9,  *py = &y ;
```



a) x =
b) *py =
c) px =
d) &y =
e) *px =
f) y =
g) py =
h) &x =
i) px++ =
j) *&px =

15) Considerando as variáveis e ponteiros definidos abaixo, quais são as atribuições permitidas?

```
int     i,  *pi, ;
float  f,  *pf;
```

a) i = f;
b) pf = &i;
c) *pf = 5.9;
d) *pi = 7.3;
e) *pf = 10;
f) f = i;
g) pi = &f;

16) Qual a saída do programa abaixo sabendo-se que a variável x está localizada no endereço 0x6dfef4 e a variável y está localizada no endereço 0x6dfef0 ?

```
int main()
{
        int x=4, y=7;
        cout << "\n &x= " << &x << "\t x=" << x;
        cout << "\n &y= " << &y << "\t y=" << y;

        int *px, *py;
        px = &x;    py = &y;
        cout << "\n px= " << px << "\t *px=" << *px;
        cout << "\n py= " << py << "\t *py=" << *py;
}
```

17) Seja a seguinte sequência de instruções em um programa C:

```
int *pti;
int veti[]={10,7,2,6,3};
pti = veti;
```

Qual afirmativa é falsa?

- a. *pti é igual a 10
- b. *(pti+2) é igual a 2
- c. pti[4] é igual a 3
- d. pti[1] é igual a 10
- e. *(veti+3) é igual a 6

18) Encontre os erros do programa e reescreva-o para que funcione corretamente:

```
#include <iostream>
//Mostra passagem de matrizes para funções como argumento
using namespace std;
float media (float lista, int tamanho);      // Declara um ponteiro variável
int main()
{
   const int MAXI=20;
   float notas[MAXI];     int i;

   for (i=0; i<=MAXI; i++)  {
      cout << "Digite a nota do aluno: " << (i+1) << ": ";       cin >> (notas+i);
      if ( *(notas+i) < 0 ) break;
   }
   int m = media (notas);
   cout << "\n\n Média das notas:  << m;
}
float media (float *lista, int tamanho)
{
   for (int i=1; i<tamanho; i++)
      m+= lista++;
   return (m/tamanho);
}
```

19) Encontre os erros do programa e reescreva-o para que funcione corretamente:

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

class venda {
private:
   int npecas; float preco;
public:
   void getvenda() {
      cout << "\n Insira o no.de pecas:";  cin  <<  npecas;
      cout << "\n Insira o preco:";           cin <<  npreco;
   }
   printvenda() const {
      cout << "\n No. de pecas:"  << npecas;
      cout << "\n Preco:" << preco;
   }
};

void  main ()  {
   Venda A;
   A.getvenda();              A.printvenda();

   Venda *B;
   B.getvenda();              B.printvenda();
}
```

Obs: outros exercícios em: https://www.codingame.com/playgrounds/24988/programacao-c/praticando-ponteiros-e-funcoes

Polimorfismo

Habilidade de adquirir várias formas. No contexto da POO, isto significa que o mesmo nome da função denota várias funções que são membros de diferentes objetos. Polimorfismo é uma ferramenta poderosa em OO. Basta enviar uma mensagem padrão para vários objetos diferentes sem especificar como ela será compreendida. Não há necessidade de conhecer de que tipo o objeto é. O recebedor é responsável por interpretar a mensagem é compreendê-la. Novas unidades também podem ser adicionadas a um programa complexo sem necessidade de recompilar o programa inteiro.

Exemplo:

```
class Classe1 {
public:
        virtual void f() {
                cout << "Função f() na Classe1\n";
        }
        void g() {
                cout << "Função g() na Classe1\n";
        }
};

class Classe2 {
public:
        virtual void f() {
                cout << "Função f() na Classe2\n";
        }
        void g() {
                cout << "Função g() na Classe2\n";
        }
};

class Classe3 {
public:
        virtual void h() {
                cout << "Função h() na Classe3\n";
        }
};

int main() {
        Classe1 objeto1, *p;
        Classe2 objeto2;
        Classe3 objeto3;
        p = &objeto1;
        p->f();                                         // Função f() na Classe1
        p->g();                                          //Função g() na Classe1
        p = (Classe1*) &objeto2;
        p->f();                                         //Função f() na Classe2
        p->g();                                         //Função g() na Classe1
        p = (Classe1*) &objeto3;
        p->f();         // Possível término anormal do programa
        p->g();                                         //Função g() na Classe1
        // p->h();      // h() não é um membro de Classe1;
        return 0;
}
```

Quando **p** é declarada como um ponteiro ao **objeto1** do tipo de classe **Classe1**, dois membros de função – definidos na **Classe1** – são ativados. Mas depos que **p** se torna um ponteiro para o **objeto2** do tipo de classe **Classe2**,

p->f() ativa a função definida na **Classe2**, enquanto que **p->g()** ativa uma função definidade da **Classe1**. A diferença se encontra no momento em que uma decisão é tomada com relação à função a ser chamada.

Na chamada **associação estática**, a decisão com relação a uma função a ser executada é determinada em tempo de compilação. Já na **associação dinâmica**, a decisão é postergada até o tempo de operação. Em C++, **a associação dinâmica é forçada declarando-se uma função-membro como virtual**. Assim, se um membro da função virtual é chamado, a função escolhida para execução depende não do tipo de ponteiro determinado por sua declaração, mas do tipo do valor que o ponteiro atualmente tem. No exemplo, o ponteiro **p** foi declarado do tipo **Classe1***. Logo, se **p** aponta para a função **g()**, que não é virtual, independentemente do local do programa no qual a instrução de chamada **p->g()** ocorre, é sempre considerado uma chamada à função **g()** definida na **Classe1**. Isto porque o compilador toma esta decisão baseado na declaração de tipo de **p** e de **g** ser não virtual. Para membros de função virtuais, a decisão é tomada durante o tempo de execução; se o membro da função é virtual, o sistema procura o tipo do valor corrente do ponteiro e invoca o membro da função apropriado. Depois da declaração inicial de **p** como sendo do tipo **Classe1***, a função virtual **f()**, pertencendo à **Classe1**, é chamada, ao passo que, depois de atribuir a **p** o endereço do **objeto2** do tipo **Classe2, f()**, pertencente à **classe2**, é chamada.

Depois que foi atribuído a **p** o endereço de **objeto3**, ainda se invoca **g()** definida na **Classe1**. Isto porque **g()** não é redefinida na **Classe3** e **g()**, a partir da **Classe1**, é chamada. Não se pode chamar **p->f()** pois **C++ escolhe a** primeira função virtual na **Classe3**, porque **f()** é declarada virtual na **Classe1**, de modo que o sistema tenta encontrar, sem sucesso, na **Classe3**, a definição de **f()**. Ainda que **p** aponte para o **objeto3**, a instrução **p->h()** resulta em erro de compilação, porque o compilador não encontra **h()** na **Classe1**, onde **Classe1*** é ainda o tipo do ponteiro **p. Para o compilador, não importa que h() seja definida na Classe3 (seja ela virtual ou não)**.