DEFINIÇÃO DE CLASSES EM C++

Linguagem Compilada Orientada a Objectos, considerada como o C da próxima geração, criada em meados dos anos 80.
Não nos debruçaremos sobre o paradigma da Programação Orientada para Objectos uma vez que já foi objecto de estudo em disciplinas anteriores. Referiremos sòmente alguns aspectos da linguagem de modo a permitir criar e manipular estruturas que serão a base dos tipos abstractos de dados que estudaremos ao longo da disciplina.

Classes

Class *nome da classe*

```
{
private:
        membros privados
protected:
        membros protegidos
public:
        membros públicos
};
```

Na declaração da classe são colocados os protótipos dos métodos, só em métodos com código reduzido é feita a definição dentro da estrutura class.

Definição de métodos fora da estrutura class:

[Tipo_retorno] nome_classe :: nome_método([parametros])

```
{
        codigo do metodo
}
```

EXEMPLO:

```
#include <iostream.h>
class circulo
{
private:
        int coordx;
        int coordy;
        int raio;
        char * cor;

public:
        circulo(); //construtor
        circulo(int x,int y,int r,char *c); //construtor
        circulo(const circulo c); //construtor cópia que se verá mais adiante
        ~circulo(); //destrutor que se verá mais adiante
        void listar();
```

```
};

circulo::circulo()
{       coordx=0;
        coordy=0;
        raio=0;
        cor=NULL;
};

circulo::circulo(int x,int y,int r,char *c)
{
        coordx=x;
        coordy=y;
        raio=r;
        cor=new char[strlen(c) +1];
        strcpy(cor,c);
};

circulo::circulo (const circulo &c)    //construtor cópia descrito mais tarde
{
        raio=c.raio;
        coordx=c.coordx;
        coordy=c.coordy;
        cor= new char  [strlen(c.cor)+1]  ;
        strcpy(cor,c.cor);
};

circulo::~circulo ()  //destrutor descrito mais tarde
{
        delete [] cor;
};

void circulo::listar()
{
        cout<<"cordenada x="<<coordx<<'\n';
        cout<<"cordenada y="<<coordy<<'\n';
        cout<<"raio="<<raio<<'\n';
        cout<<"cor="<<cor<<'\n';
};

void main()
{
        circulo c1;
        circulo c2(10,15,33,"Azul");
        c1.listar();
        c2.listar();
}
```

FUNÇÕES e PARÂMETROS

int f1(int x) x parâmetro valor
int f2(int& y) y parâmetro referência
int f3(const int& z) z parâmetro referência constante

EXEMPLO:

```
#include <iostream.h>
#include<string.h>
class circulo
{
private:
        int coordx;
        int coordy;
        int raio;
        char *cor;
public:
        circulo();
        circulo(int x,int y,int r,char *c);
        ...  ...  ...
        void listar();
        void testaParam1(int a);
        void testaParam2(int& a);
        void testaParam3(const int& a);
};

circulo::circulo()
{
        coordx=0;
        coordy=0;
        raio=0;
        cor=NULL;
};

circulo::circulo(int x,int y,int r,char *c)
{
        coordx=x;
        coordy=y;
        raio=r;
        cor=new char[strlen(c) +1];
        strcpy(cor,c);
};

void circulo::listar()
{
        cout<<"coordenada x="<<coordx<<'\n';
        cout<<"coordenada y="<<coordy<<'\n';
        cout<<"raio="<<raio<<'\n';
        cout<<"cor="<<cor<<'\n';
};
```

```
void circulo::testaParam1(int a)
{
        a=11;
        raio=a;
};

void circulo::testaParam2(int& a)
{
        a=22;
        raio=a;
};

void circulo::testaParam3(const int& a)
{
        //a=33;  Erro, constante não admite atribuição
        raio=a;
};

void main()
{
        int i;
        circulo c(10,15,100,"Amarelo");
        c.listar();
        i=4;
        c.testaParam1(i);
        cout<<"i="<<i<<'\n';
        c.listar();
        c.testaParam2(i);
        cout<<"i="<<i<<'\n';
        c.listar();
        c.testaParam3(i);
        cout<<"i="<<i<<'\n';
        c.listar();
}
```

RESULTADOS

```
coordenada x=10
coordenada y=15
raio=100
cor=Amarelo

i=4     // i não foi alterado – passagem por valor
cordenada x=10
coordenada y=15
raio=11
cor=Amarelo

i=22    // i foi alterado – passagem por referência
```

coordenada x=10
coordenada y=15
raio=22
cor=Amarelo

i=22 // i não admitiu alteração dentro do método
coordenada x=10
coordenada y=15
raio=22
cor=Amarelo

PARÂMETROS VALOR
Sempre que há uma passagem de parâmetros por valor é executado o construtor cópia do tipo de dados do parâmetro formal.

Quando uma função termina destrutores dos tipos de dados dos parâmetros formais "destroem" os valores desses parâmetros.

PARÂMETROS REFERÊNCIA
Os parâmetros actuais são ligados aos parâmetros formais, não existe cópia dos parâmetros, como no caso anterior. O nome dos parâmetros formais substitui o nome dos parâmetros actuais. Poupa-se tempo. Qualquer alteração do parâmetro formal, altera o parâmetro actual.

PARÂMETROS REFERÊNCIA CONSTANTE
Neste caso o parâmetro formal substitui o actual ,mas como é constante, não admite alteração de valor.

Nota: Se não pretendemos alterar o valor do parâmetro actual usamos uma passagem por valor no caso de tipos de dados primitivos, no caso de outros tipos de dados dever-se-á usar referência constante.

VALORES DE RETORNO

Os valores de retorno podem ser do tipo **valor**, **referência** e **referência constante**

No caso dos valores de retorno, se for devolvido um valor (objecto), aquilo que é devolvido é copiado no ambiente de retorno, recorrendo-se ao construtor cópia.

Exemplo1:
```
int circulo::retornaRaio()
{
        return raio;
};
```

Por questões de eficiência, no caso da devolução de tipos não primitivos, não

devemos usar o retorno por valor mas o retorno de referência, evitando-se a invocação do construtor cópia

Exemplo2:

```
int& circulo::retornaRaio()
{
        return raio;
};
```

NOTA1:Só podemos devolver referências de objectos globais à função.

NOTA2:Há um alias entre a função e o objecto que ela devolve, no exemplo anterior podemos alterar o valor do raio invocando retornaRaio()++.

Se o valor a devolver for uma referência constante já não admite a invocação anterior.

EXEMPLO3

```
const int& circulo::retornaRaio()
{
        return raio;
};
```

EXEMPLO4

```
class pequena
{
private:
        int x;
public:
        pequena();
        pequena(int a);
        void listar();
        int  & obterx();
        int& obtermembro();
};


pequena::pequena()
{
        x=5;
};

pequena::pequena(int a)
{
        x=a;
};

void pequena::listar()
{
```

```
        cout<<"x="<<x<<'\n';
};

 int& pequena::obter2x()
{
        x=2*x;
        return x;
};

int& pequena::obtermembro()
{
        return x;
}

void main()
{
        int i;
        pequena p(45);
        p.listar();
        const int& y=p.obter2x();
        cout<<"y="<<y<<'\n';
//y++;Erro como a referência y é constante não admite atribuição
        cout<<p.obtermembro()<<'\n';
        i=p.obter2x();
        cout<<"y="<<y<<'\n';
// Se obter2x()devolvesse uma referencia constante isto não era possivel
        p.obter2x()++
        p.listar();
}
```

RESULTADOS:

x=45
Y=90
90
Y=180
x=361

FUNÇÔES CONSTANTES

Métodos constantes não permitem que sejam alterados os objectos que os invocam.

```
Void circulo::alteraRaio() const
{
raio=88; //   ERRO,estamos a alterar um membro de dados do objecto circulo que
invocou o método
};
```

CONSTRUTORES

1. Sem Parâmetros circulo();
2. Com Parâmetros e Parâmetros por defeito
   circulo(int x,int y,int r,char *c)
   circulo(int x=0,int y=0,int r=5,char * c="azul")
   Os valores por defeito (0,0,5,azul) só são indicados na declaração do método.

3. Cópia circulo(const circulo& c)

Tem a mesma designação da classe
Não devolvem quaisquer valores

NOTA: Temos que implementar o construtor cópia sempre que na classe existem membros de dados não primitivos ou que exigem alocação dinâmica de memória.

CONSTRUTOR CÓPIA

Este construtor caracteriza-se por ter um só parâmetro, parâmetro esse que é do mesma classe a que pertence o construtor ou em que o 1º parâmetro (no caso de ter mais do que um parâmetro) é do mesmo tipo que a classe.

Quando se passa um objecto por valor a uma função o compilador invoca de forma automática o construtor cópia.
Assim, na construção de construtores cópia não podemos passar o objecto, senão cairíamos numa recursividade infinita, logo há que passar uma referência para o objecto. Se tal não fizermos o compilador de C++ assinala erro. Atendendo a que não se pretende alterar o objecto que é passado por referência essa referência deverá ser constante

Quando uma função devolve um objecto também é invocado o construtor cópia.
Se passarmos e devolvermos referências para objectos em vez dos próprios objectos eliminamos o "overhead" de invocação do construtor cópia.

Se o construtor cópia não estiver definido, sempre que numa função há a passagem ou retorno de um objecto , o compilador gera um construtor cópia automàticamente, criando um novo objecto através de uma operação de cópia membro a membro do objecto que é passado. Isto tem problemas sempre que o objecto tem um atributo que exige alocação de memória porque são igualados apontadores, passando a ter a mesma área de memória referenciada por dois apontadores pertencentes a objectos distintos.

Exemplo: Voltemos à classe circulo

```
//construtor cópia
circulo::circulo (const circulo &c)
        {
                raio=c.raio;
                coordx=c.coordx;
                coordy=c.coordy;
                cor= new char  [strlen(c.cor)+1]  ;
                strcpy(cor,c.cor);
```

```
};
```

Este tipo de construtor não seria gerado automàticamente, há necessidade do programador o construir, porque na geração automática seriam igualados os apontadores cor e c.cor.

Dispondo de construtor cópia, podemos criar um objecto à custa de outro já criado, do seguinte modo:

```
circulo circ1(8,5,30,"rosa");
circulo circ2(circ1);
```

Assim circ2 é um círculo inicializado com os mesmos dados de circ1, para tal recorreu ao construtor cópia.

Um objecto pode ainda ser criado dinamicamente através do operador **new** que além de alocar espaço na memória invoca o respectivo construtor.

Assim, poderíamos ter o seguinte código:

```
circulo * apcirc1;
apcirc1=new circulo(10,11,12,"verde"); //cria o objecto e invoca o construtor com parâmetros
apcirc1->listar();
                OU
circulo *apcic1= new circulo(10,11,12,"verde");
```

Para libertar o espaço ocupado pele objecto far-se-ia **delete** apcirc1, que automaticamente invocaria o destrutor de circulo

DESTRUTORES

1. Tem a mesma designação da classe precedidos de ~
   ```
   ~circulo();
   ```
2. Não tem parâmetros
3. Não devolvem quaisquer valores
4. Não são invocados explicitamente.
5. São invocados automaticamente sempre que termina o âmbito de um objecto da classe ou que é feito delete de um ponteiro para um objecto da classe.
6. É necessário implementar os destrutores sempre que os membros de dados fazem alocação dinâmica de memória, para libertarem o espaço alocado por esses membros.

No exemplo do círculo o destrutor teria o seguinte código:

```
circulo::~circulo()
{
delete [ ] cor;
}
```

Neste programa considerou-se ainda implementado no destrutor a libertação do ponteiro cor. Quando o âmbito de um objecto finaliza é invocado o destrutor e assim é libertado o espaço de memória, que tinha sido alocado para a cor através de new no respectivo construtor.

SOBRECARGA DE OPERADORES

1. Permite reescrever codigo para operadores de modo a manipularem operandos de classe diferente da que estão predefinidos.
2. Não é possível alterar a precedência nem o número de operandos.

EXEMPLO: Sobrecarregar o operador + para a classe circulo, supondo que o circulo resultado, mantem a origem do primeiro operando e o raio é a soma dos raios.

```
Circulo circulo::operator +( const circulo& c)
        {       circulo temp;
                temp.coordx=this->coordx;
                temp.coordy=this->coordy;
                temp.raio=this->raio + c.raio;
                return temp;
}
```

Nas aulas práticas serão implementadas outras classes e feitos outros operadores, como o de atribuição, os de comparação,...

Há operadores que não admitem sobrecarga, tal é ocaso de :

. :: ?: #

Quando um operador é definido como método de uma classe, tem menos um parâmetro do que se fosse definido como uma função normal fora da classe. Isto porque o primeiro parâmetro de um operador definido como função membro de uma classe é o objecto que invoca o método,que é passado como parâmetro implícito. No exemplo anterior, soma de dois círculos, só consideramos um parâmetro correspondente ao 2º operando do operdor +, o primeiro operando será o objecto referenciado por this.