

# SuperComputação

## Aula 2 – C++ (Orientação a Objetos)

2019 – Engenharia

Luciano Soares [<lpsoares@insper.edu.br>](mailto:lpsoares@insper.edu.br)

Igor Montagner [<igorsm1@insper.edu.br>](mailto:igorsm1@insper.edu.br)

# Parte 2 – C++ OO

# Parâmetros padrões em funções

- Em C ++ as funções podem ter parâmetros padrões, para os quais não são necessários argumentos na chamada, de tal forma que, por exemplo, uma função com três parâmetros possa ser chamada com apenas dois. Exemplo:

```
int divide (int a, int b=2)
{
    int r;
    r=a/b;
    return (r);
}
```

# Overload de Funções

- Em C ++, funções podem ter o mesmo nome se seus parâmetros forem diferentes. Isso não funciona se só o retorno da função for diferente.

```
int sum (int a, int b)
{
    return a+b;
}
```

```
double sum (double a, double b)
{
    return a+b;
}
```

# Templates - Criação

- Os *templates* são a base da programação genérica, que envolve escrever código de maneira independente de qualquer tipo específico.  
Exemplo:

```
template <class T>  
T sum (T a, T b)  
{  
    return a+b;  
}
```

# Templates - Invocação

- Para chamar uma função com *template* use o tipo que deseja entre os sinais de menor e maior.

Exemplo:

```
x = sum<int>(10,20);
```

O compilador pode deduzir os tipos algumas vezes, contudo a recomendação é especificar o tipo nas chamadas.

# Declarando variáveis automaticamente

- Possibilidade de buscar o tipo de variável já usada para declarar uma nova, exemplo:

```
int foo = 0;  
auto bar = foo; // o mesmo que: int bar = foo;
```

- Outra opção

```
int foo = 0;  
decltype(foo) bar; // o mesmo que: int bar;
```

auto e decltype possuem diferenças, preste atenção quando for usar.

# Explicit type casting

- Em C ++ é possível usar a notação funcional precedendo a expressão a ter o tipo de dado convertida, exemplo:

```
float f = 10.0;
```

```
int i = int (f);
```



# Library arrays

- Permite usar recursos “avançados” em vetores. Como tamanho dinâmica, inserção e remoção, etc.

```
#include <iostream>
#include <array>
int main(){
    array<int,3> myarray = {10,20,30};
    for (int i=0; i<myarray.size(); ++i)
        ++myarray[i];
}
```

# For sobre faixa de valores

Executa um loop sobre uma faixa de valores.

A sintaxe é:

```
for ( declaração : faixa ) statement;
```

Exemplo:

```
array<int,3> myarray = {10,20,30};
```

```
for (int elem : myarray)  
    cout << elem << '\n';
```

# Substituir a declaração da variável por auto.

```
std::string str {"Hello!"};  
for (char c : str)  
{  
    std::cout << "[" << c << "];"  
}  
std::cout << '\n';
```

# Ponteiros Nulos

- Tradicionalmente (em C) se usa o valor 0 (zero). Contudo C++ traz o recurso do nullptr que é recomendado por ser mais seguro em certas situações.

Exemplo de uso:

```
int* p = nullptr;
```

# Enum Class

Em C ++ é possível criar tipos de enum que não são implicitamente conversíveis em int, mas do próprio tipo enum. Exemplo:

```
enum class Cores {preto, azul, verde, amarelo};
```

# O que acontece com esse código?

```
const char foo[] = "ola pessoal";  
foo[1] = 'i' ;  
*(foo+2) = 'e';  
std::cout << foo << std::endl;
```

# Esse código está correto?

```
char a;  
char * b;  
char ** c;  
a = 'z';  
b = &a;  
c = &b;
```

# Argumentos passados por valor, por referência e ponteiros

Lembra dessa função, e agora essa função abaixo ira se comportar como esperado?

```
void duplicate(int a, int &b, int *c) {  
    a *= 2;  
    b *= 2;  
    *c *= 2;  
}
```



# Structs Alocadas Dinamicamente

Para acessar um membro de uma variável alocada dinamicamente, você pode usar os símbolos -> no lugar o ponto. Por exemplo:

```
struct xpto {  
    int a;  
    float b;  
};  
xpto *abc = new xpto;  
abc->a = 30;  
std::cout << abc->a << std::endl;
```

Na prática o acesso também poderia ser:

```
(*xpto).a
```

# Orientação a Objetos

Classes são estruturas que organizam dados e funcionalidades em uma única estrutura. A sintaxe de uma classe é:

```
class class_name {  
    access_specifier_1: member1;  
    access_specifier_2: member2;  
    ...  
} object_names;
```

# Especificadores de Acesso

As classes podem definir o acesso a seus membros e métodos através dos especificadores de acesso, que são eles:

**private:** os membros privados de uma classe são acessíveis apenas dentro de outros membros da mesma classe (ou de seus "amigos").

**protected:** membros protegidos são acessíveis a partir de membros da mesma classe, mas também de membros de suas classes derivadas.

**public:** os membros públicos são acessíveis de qualquer lugar onde o objeto esteja visível.

# Exemplo de Classe

```
class Rectangle {  
    int width, height;  
    public:  
        void set_values(int,int);  
        int area(void);  
} rect;
```

Por padrão os membros de uma classe são private.

O que é o rect nesse exemplo acima?

# Implementando Funcionalidades

As funcionalidades dos métodos de uma classe podem ser implementados diretamente na definição da classe ou fora usando os dois dois pontos.

Exemplo:

```
class Rectangle {  
    int width, height;  
public:  
    void set_values(int,int);  
    int area() {return width*height;}  
};  
void Rectangle::set_values(int x, int y) {  
    width = x; height = y;  
}
```

# Instanciando Objetos

(não dinamicamente)

Depois de definida a classe, objetos podem ser instanciados a partir dela, e os membros e métodos acessados pelo operador ponto (.).

Por exemplo:

```
Rectangle retangulo;  
retangulo.set_values(3,4);  
int myarea = retangulo.area();
```

# Construtores

Construtores podem ser definidos em classes. Eles serão como métodos, porém com o mesmo nome da classe e sem nenhum tipo de retorno (nem void).

Parâmetros adicionais podem ser definidos em um construtor. Por exemplo:

```
class Rectangle {  
    int width, height;  
    public:  
        Rectangle (int,int);  
        int area () {return (width*height);}  
};  
Rectangle::Rectangle (int a, int b) {  
    width = a;  
    height = b;  
}
```

# Instanciando com construtores

Agora ao instanciar uma classe você deve passar os parâmetros do construtor. Por exemplo

```
Rectangle rect (3,4);  
Rectangle rectb (5,6);
```



# Criando construtores

Construtores de classes podem usar recursos especiais para iniciar variáveis membros diretamente. Por exemplo, uma construção assim:

```
Rectangle::Rectangle (int x, int y) { width=x; height=y; }
```

Poderia ser feita assim:

```
Rectangle::Rectangle (int x, int y) : width(x), height(y) { }
```

# Objetos Instanciados Dinamicamente

É possível criar objeto dinamicamente usando os recursos de ponteiros e alocação de memória do C++. Para acessar atributos e métodos se deve usar o operador de flecha (->). Exemplo:

```
Rectangle *foo;  
foo = new Rectangle (5, 6);  
cout << "area= " << foo->area() << '\n';  
delete foo;
```

# Sobrecarga (overloading) de Operações

Em classes é possível usar as operações básicas da linguagem C++ sobrecarregando funções com nomes específico. As operações permitidas são:

Overloadable operators												
+	-	*	/	=	<	>	+=	--	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	
~	&=	^=	=	&&		%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

Veja o exemplo no próximo slide.

# Exemplo de sobrecarga de operadores

```
class CVector {  
    public:  
        int x,y;  
        CVector () {};  
        CVector (int a,int b) : x(a), y(b) {}  
        CVector operator+ (const CVector&);  
};
```

```
CVector CVector::operator+ (const CVector& param) {  
    CVector temp;  
    temp.x = x + param.x;  
    temp.y = y + param.y;  
    return temp;  
}
```

```
int main () {  
    CVector foo (3,1);  
    CVector bar (1,2);  
    CVector result;  
    result = foo + bar;  
    cout << result.x << ',' << result.y << '\n';  
    return 0;  
}
```

# Outra Revisão (em parte)

**this:** ponteiro para o próprio objeto em uso.

**static:** atributo (variável) compartilhada por todos os objetos, chamada de variável da classe método que só acessa atributos (variáveis) estáticas e outros métodos estáticos da classe

**const:** permite métodos retornarem referências consts (somente leitura).

# Templates em Classes

Classes podem usar *templates* para suportar diferentes tipos de dados em seu uso. Por exemplo:

```
template <class T>
class mypair {
    T values [2];
public:
    mypair (T first, T second)
    {
        values[0]=first; values[1]=second;
    }
};
```

# Destrutores

Destrutores são métodos invocados quando desejamos parar de usar uma classe. C++ não tem um sistema de “garbage collection” assim se alocarmos memória dinâmica precisamos liberar e os destrutores são um local. Os destrutores tem o nome da classe com um til (~) na frente. Exemplo:

```
class Example {  
    string* ptr;  
public:  
    Example(): ptr(new string){}  
    ~Example() {delete ptr;}  
};
```

# Herança

Uma classe pode herdar seus atributos e métodos para uma classe mais especializada. A herança pode ser pública, protegida ou privada.

Para fazer isso, depois do nome na nova classe, coloque dois pontos, o tipo de herança e o nome da classe à herdar.



# Exemplo de Herança

```
class Polygon {  
    protected:  
        int width, height;  
    public:  
        void set_values (int a, int b) { width=a; height=b;}  
};
```

```
class Rectangle: public Polygon {  
    public:  
        int area () { return width * height; }  
};
```

```
class Triangle: public Polygon {  
    public:  
        int area() { return width * height / 2; }  
};
```

# Polimorfismo

Um dos principais recursos de herança de classe é que um ponteiro para uma classe derivada é compatível com o tipo de ponteiro para sua classe base.

Veja o exemplo a seguir.

# Exemplo de Polimorfismo

```
class Polygon {
protected:
    int width, height;
public:
    void set_values (int a, int b)
        { width=a; height=b; }
};

class Rectangle: public Polygon {
public:
    int area() { return width*height; }
};

class Triangle: public Polygon {
public:
    int area() { return
width*height/2; }
};
```

```
int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon *ppoly1 = &rect;
    Polygon *ppoly2 = &trgl;
    ppoly1->set_values(4,5);
    ppoly2->set_values(4,5);
    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```

# Métodos Virtual

Um método virtual permite que um método de uma classe derivada com o mesmo nome seja adequadamente chamado de um ponteiro da classe base.

Veja o exemplo a seguir.

# Exemplo de Método Virtual

```
class Polygon {  
protected:  
    int width, height;  
public:  
    void set_values (int a, int b)  
        { width=a; height=b; }  
    virtual int area () { return 0; }  
};  
  
class Rectangle: public Polygon {  
public:  
    int area () { return width * height; }  
};  
  
class Triangle: public Polygon {  
public:  
    int area () { return (width * height / 2); }  
};
```

```
int main () {  
    Rectangle rect;  
    Triangle trgl;  
    Polygon poly;  
    Polygon * ppoly1 = &rect;  
    Polygon * ppoly2 = &trgl;  
    Polygon * ppoly3 = &poly;  
    ppoly1->set_values (4,5);  
    ppoly2->set_values (4,5);  
    ppoly3->set_values (4,5);  
    cout << ppoly1->area() << '\n';  
    cout << ppoly2->area() << '\n';  
    cout << ppoly3->area() << '\n';  
    return 0;  
}
```

# Classes Abstratas

As classes só podem ser usadas como classes base e, portanto, podem ter métodos sem nenhuma definição (conhecidos como métodos virtuais puros). A sintaxe para isso é definir o método com um = 0 (um sinal de igual e um zero).

```
class Polygon {  
    protected:  
        int width, height;  
    public:  
        void set_values (int a, int b)  
        { width=a; height=b; }  
        virtual int area () =0;  
};
```

# Só para saber que existe

- Cópias e Realocação em objetos: O C++ possui recursos automáticos para copiar e mover dados de um objeto para outro.
- friend: funções e classes podem ser amigas (friend), nesse caso elas conseguem ver atributos privados ou protegidos.
- Herança múltipla: uma classe que herda de duas ou mais classes ao mesmo tempo.



# Para casa

## Estudar:

Exceptions:

<http://www.cplusplus.com/doc/tutorial/exceptions/>

I/O (entrada e saída de arquivos):

<http://www.cplusplus.com/doc/tutorial/files/>



# Atividades

**Atividade prática:**

<https://github.com/Insper/supercomp/wiki>

**Entrega:** 20/08

**Forma:** projeto do Github com um commit para cada aula

# Referências

- Livros:

- Hager, G. ; Wellein, G. **Introduction to High Performance Computing for Scientists and Engineers**. 1ª Ed. CRC Press, 2010.

- Artigos:


- Bjarne Stroustrup, “An Overview of the C++ programming language”, THE HANDBOOK OF OBJECT TECHNOLOGY (EDITOR: SABA ZAMIR). CRC PRESS LLC, BOCA RATON. 1999

- Internet:

- <http://www.cplusplus.com/doc/tutorial/>

# Inspire

[www.inspire.edubr](http://www.inspire.edubr)



```
Circle bar = 20.0; //  
assignment objects  
Circle baz {30.0}; // uniform  
init.  
Circle qux = {40.0}; // POD-like
```

# Non-type template arguments

O segundo argumento do modelo de função `fixed_multiply` é do tipo `int`. Parece apenas um parâmetro de função regular e pode ser usado como um.

```
// template arguments
#include <iostream>
using namespace std;
```

```
template <class T, int N>
T fixed_multiply (T val)
{
    return val * N;
}
```

```
int main() {
    std::cout << fixed_multiply<int,2>(10) << '\n';
    std::cout << fixed_multiply<int,3>(10) << '\n';
}
```

Existe uma grande diferença: o valor dos parâmetros do modelo é determinado em tempo de compilação para gerar uma instancição diferente da função `fixed_multiply`, e assim o valor desse argumento nunca é passado durante o **Insper**

## Arrays as parameters

Em uma declaração de função, também é possível incluir matrizes multidimensionais. O formato de um parâmetro de matriz tridimensional é:

```
base_type[][depth][depth]
```

Por exemplo, uma função com uma matriz multidimensional como argumento poderia ser:

```
void procedure (int myarray[][3][4])
```

## Unions

```
union type_name {  
    member_type1 member_name1;  
    member_type2 member_name2;  
    member_type3 member_name3;  
    .  
    .  
} object_names;
```

```
union mytypes_t {  
    char c;  
    int i;  
    float f;  
} mytypes;
```

Cada um desses membros é de um tipo de dados diferente. Mas como todos eles estão se referindo ao mesmo local na memória, a modificação de um dos membros afetará o valor de todos eles. Não é possível armazenar diferentes valores neles de forma que cada um seja independente dos outros.