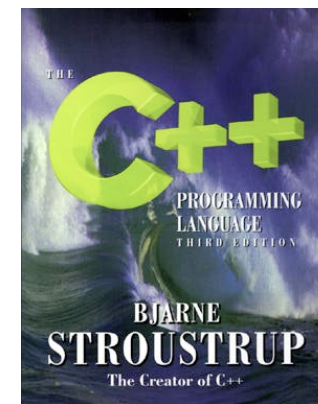


A linguagem C++

Visão comparativa face a Java



Evolução

- Desenvolvida por Bjarne Stroustrup nos AT&T Bell Labs em 1980.
- Influências: C, Simula 67, Algol 68
- Primeira versão em 1980: "C with Classes"
 - *Classes: An Abstract Data Type Facility for the C Language*, ACM SIGPLAN Notices, 1982
 - *Adding Classes to C: An Exercise in Language Evolution*, Software -- Practice and Experience, 1983
- Nome definitivo (C++) em 1983/84
- *Educational Release* em 1983
- *General Releases* 1985, 1986, 1987, 1988, 1989, 1990, ...
- ANSI/ISO C++ Standard (finalizado em 1998).

Bibliografia de Referência

- *The C++ Programming Language*, (First Edition)
Bjarne Stroustrup, Addison-Wesley, 1986
 - (desatualizado)
- *The C++ Programming Language*, (Third Edition)
Bjarne Stroustrup, Addison-Wesley, 1997.
 - A “Bíblia” actual de C++
- *The Annotated C++ Reference Manual (ARM)*,
Margaret Ellis and Bjarne Stroustrup, Addison-Wesley, 1990
 - Livro de referência, serviu de base à norma ANSI/ISO C++
- *The Design and Evolution of C++*, Bjarne Stroustrup, Addison-Wesley, 1994.

Bibliografia adicional

- Programação com Classes em C++, Pedro Guerreiro, 2000, FCA.
- Thinking in C++, 2nd ed., Volume 1, Bruce Eckel, 2000, Prentice Hall.
- C++ Primer, 3rd edition, Stanley B. Lippman, Josee Lajoie, Addison-Wesley.
- C++ : How to program, Harvey Deitel, 2001.

Semelhanças com Java

- Primitive data types
 - (in Java, platform independent)
- Syntax
 - control structures, exceptions ...
- Classes, visibility declarations (public, private)
- Multiple constructors, this, new
- Types, type casting
 - safe in Java, not in C++
- Comments
 - Not Javadocs

"Hello World" in Java

```
package p2;  
  
// My first Java program!  
  
public class HelloMain {  
    public static void main(String[] args) {  
        System.out.println("hello world!");  
    }  
}
```

"Hello World" in C++

Use the standard namespace

Include standard
iostream classes

A C++ comment

cout is an
instance of
ostream

```
using namespace std;  
#include <iostream>  
// My first C++ program!  
int main(void)  
{  
    cout << "hello world!" << endl;  
    return 0;  
}
```

operator overloading
(two *different* argument types!)

Makefiles / Managed Make in CDT

You could compile it all together by hand:

```
gcc helloWorld.cpp -o helloWorld
```

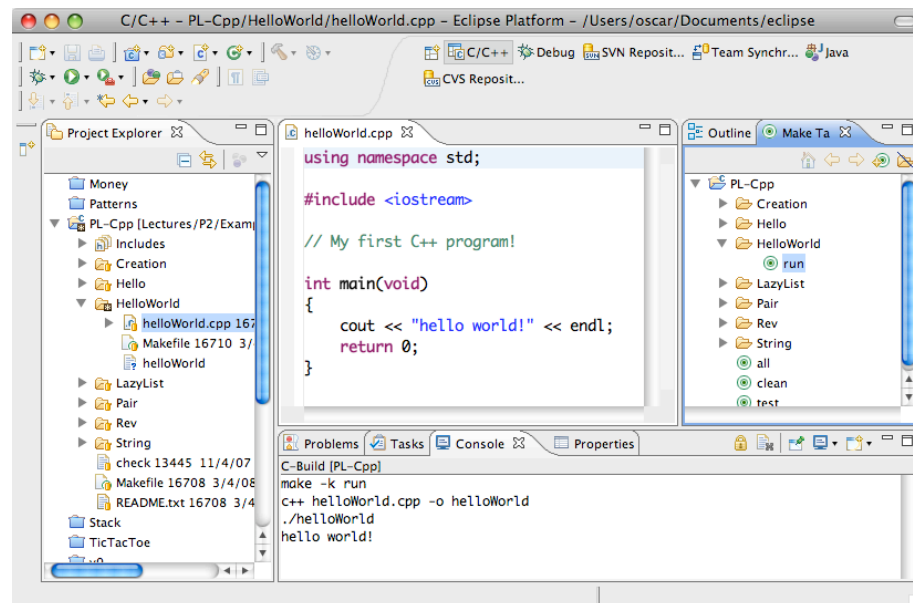
Or you could use a *Makefile* to manage dependencies:

```
helloWorld : helloWorld.cpp
```

```
gcc $@.cpp -o $@
```

```
make helloWorld
```

Or you could use *cdt with eclipse* to create a standard managed make project



namespaces

- Conflito de nomes
- Problemas de localização

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

▪ ex:

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

```
int main() {  
    int number;  
    cout << "Enter a decimal number: ";  
    cin >> number;  
    cout << "value in octal = 0"  
        << oct << number << endl;  
    cout << "value in hex = 0x"  
        << hex << number << endl;  
}
```

Tipos

- Tipos básicos
char, int, float, double
- Modificadores de tamanho
long, short, signed, unsigned
- Exemplo

```
char c;  
unsigned char c2;  
int i;  
unsigned int i2;  
short int is;  
short iis; // Same as short int  
unsigned short int isu;  
unsigned short iisu;  
long int il;  
long iil; // Same as long int  
unsigned long int ilu;  
unsigned long iilu;  
float f;  
double d;  
long double ld;
```

Ponteiros e Arrays

- vectores

```
char a[10]; // 10 caracteres (a[0] .. a[9])
```

- ponteiros

```
char *p;
```

```
int a = 47;  
int* ipa = &a;  
*ipa = 20; // a ← 20
```

- endereços

```
p = &a[3];
```

- referências

```
int num;  
int &rn = num;
```

Ponteiros

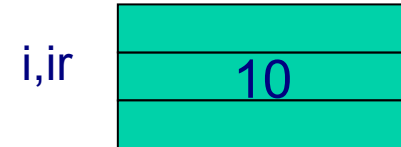
```
int i;  
int *iPtr; // a pointer to an integer  
  
iPtr = &i; // iPtr contains the address of I  
*iPtr = 100;
```

variable	value	Address in hex
	...	
i	100	456FD4
iPtr	456FD4	456FD0
	...	

Referências

A reference is an **alias** for another variable:

```
int i = 10;  
int &ir = i;    // reference (alias)  
ir = ir + 1;    // increment i
```



Once initialized, references cannot be changed.

References are especially useful in **procedure calls** to avoid the overhead of passing arguments by value, without the clutter of explicit pointer dereferencing (`y = *ptr;`)

```
void refInc(int &n)  
{  
    n = n+1; // increment the variable n refers to  
}
```

Funções: passagem de parâmetros

- Os parâmetros podem ser passados **por valor**,

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

```
...  
cout << soma(i, j);    // sem alteração de i e j
```

- .. **por ponteiro**

```
void swap(int* p, int* q) {  
    int t = *p; *p = *q; *q = t;  
}
```

```
...  
swap(&i, &j);    // invocação da função swap
```

- .. ou **por referência**.

```
void swap(int& p, int& q) {  
    int t = p; p = q; q = t;  
}
```

```
...  
swap(i, j);    // invocação da função swap
```

Templates

- Tabela de Ts (com T definido da instanciação):

```
template<class T> class Vector { // Vector de Ts
    T* v;                        // ponteiro para inteiros
    int sz;                      // total de elementos
public:
    Vector(int);                 // construtor
    ~Vector();                   // destrutor
    T& operator[] (int i);      // operador de index
};

int main() {
    Vector<int> v1(100);          // 100 inteiros
    Vector<Pessoa> v2(200);      // 200 Pessoas
    // ...
}
```

Organização de um projecto

- Ficheiros declarativos
 - (".h")
- Ficheiros de código
 - (".C" ".cc" ".cpp" ".cxx")
- Directivas de pré-processamento
 - (#include, #ifdef, #ifndef, #define ...)
- Bibliotecas de Classes
- Compilador
- Linker
- Debugger

Exemplo de um programa em C++

- Declaração de ficheiros .h (declarações)
- `int main(int argc, char **argv)`
- Instruções de entrada e saída de dados

```
// Programa que lê um número e mostra o seu inverso
#include <iostream>
using namespace std;

int main() {
    float x;
    cout << "Qual é o número?";
    cin >> x;      // leitura de x;
    cout << "O inverso de " << x << " é " << 1/x << endl;
}
```

Métodos de uma Classe

- Os métodos de uma classe são declarados dentro do corpo da classe (ficheiros .h)

```
class Screen {  
public:  
    void home();  
    void move(int, int);  
    char get();  
    char get(int, int);  
};
```

Screen.h

- A definição no corpo da classe (inline implícito).

```
class Screen {  
public:  
    void home()      { cursor = screen; }  
    char get()       { return *cursor; }  
};
```

Screen.cpp

Métodos de uma Classe

- A generalidade das definições são efectuadas fora da classe (ficheiros .cpp)

```
void Screen::home() {  
    cursor = screen;  
}
```

```
char Screen::get() {  
    return *cursor;  
}
```

Screen.h

Screen.cpp

Exemplo de uma classe

```
class Point {  
private:  
    double x;  
    double y;  
public:  
    Point(double x, double y);  
    virtual ~Point();           // not needed ..  
    virtual double getX() const;  
    virtual double getY() const;  
    virtual double distanceTo(const Point& p) const;  
    virtual void display() const;  
};
```

Point.h

Exemplo de uma classe

Point.cpp

```
Point::Point(double ix, double iy): x(ix), y(iy) {  
    }  
  
void Point::display() const {  
    cout << m_x << ", " << m_y << endl;  
}  
...
```

Objectos de uma Classe

- A definição de uma classe não se traduz em qualquer reserva de memória. Esta só é efectuada na definição de objectos da classe.

```
Screen myScreen;                // reserva estática
Screen *otherScreen = new Screen(); // dinâmica;
Screen *ptr;
ptr = &myScreen;
ptr = otherScreen;
```

- Fora do contexto da classe cada membro é identificado conjuntamente com a referência ao objecto.

```
otherScreen->get();    // "->" em ponteiros
myScreen.get();        // "." em referências
```

Construtores

```
class String {  
public:  
    String() { len = 0; str = 0; } // default constructor  
    String(const char*);           // String("Olá");  
private:  
    int len;  
    char *str;  
};  
String::String( const char *s ) {  
    len = strlen( s );  
    str = new char[ len + 1 ];  
    strcpy( str, s );  
}  
  
String st1;  
String *st2 = new String();  
String *st3 = new String("Livro");  
  
String st4("Forma estática");  
String *st5 = new String("reserva dinâmica");
```

Destrutor

- Mecanismo complementar ao construtor para "limpeza" de objectos.
- Cada classe tem um destrutor.
- É invocado (implicitamente) pelo operador delete ou quando um objecto sai fora de contexto.

```
class String {  
public:  
    virtual ~String();    // destructor  
    // ...  
};  
  
String::~~String() { delete [] str; }
```


Construtor de cópia

- Quando um objecto é inicializado com outro objecto da mesma classe é invocado um construtor especial
 - é criado pelo compilador se não for definido
 - tem a forma genérica `X::X(const X&)`.
- Esta situação pode, no entanto, conduzir a erros sempre que a classe possua objectos membros ou reserva dinâmica.
 - Neste caso os construtores dos membros não são invocados permitindo que múltiplos objectos apontem para uma mesma área de memória.

Construtor de cópia

- Exemplo

```
String nome1("Lopes");  
String nome2(nome1);
```

- o resultado será:

```
// nome2.len=nome1.len;  
// nome2.str=nome1.str; // não desejável!
```

- A resolução deste problema passa pela definição explícita de um construtor (memberwise).

```
String::String(const String& s) {  
    len = s.len;  
    str = new char[s.len+1];  
    strcpy(str, s.str);  
}
```

Utilização do Construtor de Cópia

- O construtor de cópia é chamado quando:
 - um objecto é instanciado a partir de um outro

```
Ponto centro(25, 25);  
Ponto mira(centro);  
Ponto outroPonto = centro;
```
 - um objecto é retornado por uma função

```
Ponto Figura::getCentro();
```
 - um objecto é passado por valor como parâmetro de uma função

```
void Figura::addPonto(Ponto newP);
```

Conversão implícita

Quando um argumento de tipo “errado” é passado a uma função o compilador procura o construtor adequado.

```
str = "hello world";
```

é implicitamente convertido em:

```
str = String("hello world");
```

Vectores de Objectos

- Como é inicializado um vector de objectos?

```
class Car
{
    public :
        Car(const char *vendedor = "Toyota", int nDoors = 4);

    private :
        char *m_vendedor;
        int m_nDoors;
};

Car::Car(const char *vendedor, int numDoors) : m_nDoors(numDoors)
{
    m_vendedor = new char[strlen(vendedor)+1];
    strcpy(m_vendedor, vendedor);
}

int main()
{
    Car frota[50];

    // This creates an array of 50 cars. The Car
    // constructor is called 50 times. All 50
    // cars have a vendedor of "Toyota", and 4
    // doors.
};
```

Organização típica de uma classe

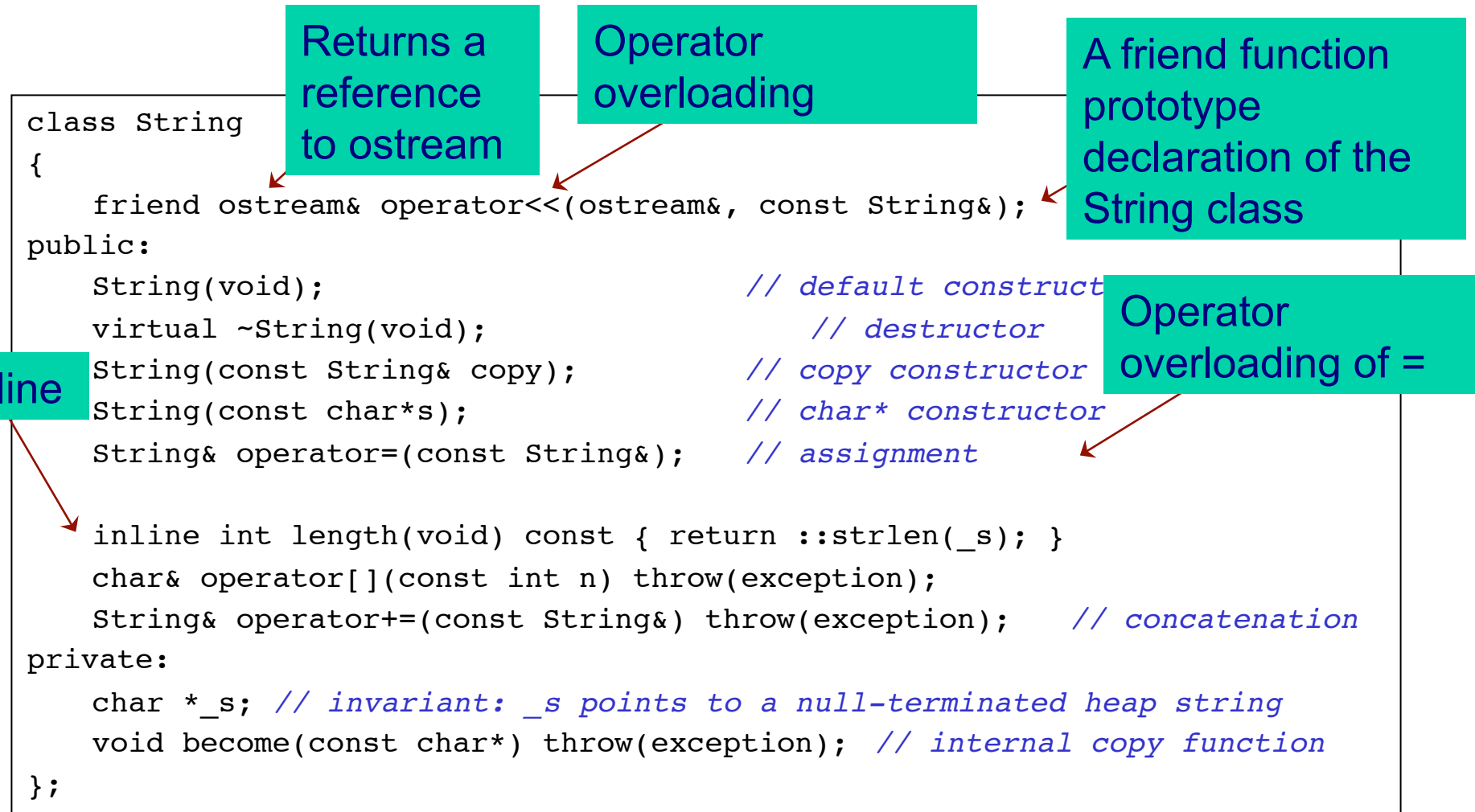
```
class myClass {  
public:  
    myClass(void);           // default constructor  
    myClass(const myClass& copy); // copy constructor  
    ...                     // other constructors  
    virtual ~myClass(void); // destructor  
    myClass& operator=(const myClass&); // assignment  
    ...                     // other public member functions  
private:  
    ...  
};
```

Organização típica de uma classe

If you don't define these four member functions, *C++ will generate them:*

- ***default constructor***
 - will call default constructor for each data member
- ***destructor***
 - will call destructor of each data member
- ***copy constructor***
 - will *shallow copy* each data member
 - pointers will be copied, not the objects pointed to!
- ***assignment***
 - will *shallow copy* each data member

A Simple String.h



Herança

Person.h

```
#include <iostream>
#include <string.h>
using namespace std;

class Person {
    public:
        Person(char* n, string bi);
        string getBI();
        char* getName();    // string getName(); <- melhor
        virtual ~Person();
    private:
        char* name;        // string name;
        string bi;
};
```

Person.cpp

```
Person::Person(char* n, string bi) {  
    name = new char[strlen(n)+1];  
    strcpy(name, n);  
    this->bi = bi;  
}  
  
char* Person::getName() { return name; }  
  
string Person::getBI() { return bi; }  
  
Person::~~Person() {  
    delete[] name;  
}
```

Herança

```
class Employee: public Person {  
    public:  
        Employee(char *name, string bi, float sal):  
            Person(name, bi), salary(sal) {}  
        float getSalary() { return salary; }  
    private:  
        float salary;  
};
```

O que é herdado?

- Todos os membros de dados não estáticos
- Todos os métodos excepto
 - construtores
 - destrutores
 - operador de atribuição
- Uma classe derivada herda de todas as classes Base que estão acima na hierarquia de herança.
 - Ao construir a hierarquia podemos controlar aquilo que queremos deixar acessível para as classes derivadas.

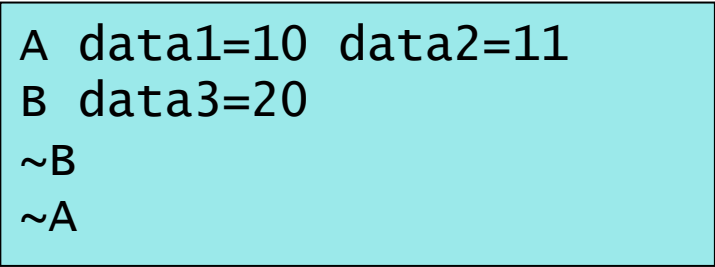
Construtores e Destrutores

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

class A {
public:
    A(int a) : data1(a), data2(a+1)
        { cout << "A data1=" << data1 << ", data2=" << data2 << endl;}
    ~A() {cout<<"~A"<<endl;}
private:
    int data1, data2;
};

class B : public A {
public:
    B(int a, int b) : A(a), data3(b) {cout << "B data3=" << data3 << endl;}
    ~B() {cout << "~B" << endl;}
private:
    int data3;
};

void main ()
{
    B umB(10,20);
}
```



```
A data1=10 data2=11
B data3=20
~B
~A
```

Default Operator =

- Se uma classe não redefinir o operador de atribuição (=) o compilador fornecerá um por defeito.
- No caso de termos uma classe derivada
 - Invocará o operador de atribuição da classe base
 - Executa *member-wise copy* na classe derivada
- No entanto se os dois existirem (base e derivado) então o derivado deverá explicitamente chamar o base

Sobrecarga do Operador de atribuição

- Se uma classe derivada redefinir o operador de atribuição deverá invocar explicitamente o operador de atribuição da classe base

```
classB& classB::operator=(const classB & rhs)
{
    if (this != &rhs)
    {
        classA::operator=(rhs);
        m_b = rhs.m_b
    }
    return *this;
};
```


Polimorfismo

Ligação estática

```
class Pessoa
{
public:
    void anunciar ()
    {
        cout << "Sou Pessoa!" << endl;
    }
};
```

```
class Estudante : public Pessoa
{
public:
    void anunciar ()
    {
        cout << "Sou Estudante!" << endl;
    }
};
```

```
void anuncio (Pessoa& p)
{
    p.anunciar();
}
```

```
int main ()
{
    Estudante a1;
    anuncio (a1);

    return 0;
}
```



Sou Pessoa!



Static binding

Ligação dinâmica

```
class Pessoa
{
public:
    virtual void anunciar ()
    {
        cout << "Sou Pessoa!" << endl;
    }
};
```

```
class Estudante : public Pessoa
{
public:
    void anunciar ()
    {
        cout << "Sou Estudante!" << endl;
    }
};
```

```
void anuncio (Pessoa& p)
{
    p.anunciar();
}
```

```
int main ()
{
    Estudante a1;
    anuncio (a1);

    return 0;
}
```



Sou Estudante!



Dynamic binding

Polimorfismo - recomendação

- Todos os métodos deverão ser descritos como métodos virtuais
 - apesar da "ligeira" degradação de desempenho permite evitar erros futuros
 - A definição de virtual basta aparecer na definição da classe base
- Um construtor não pode ser virtual
 - Necessita informação acerca do tipo exacto do objecto a ser criado
 - Não pode ser invocado para um objecto já existente.
 - Não podemos ter um ponteiro para um construtor.

Ponteiro this, declaração friend

this

- Em C++ todos os métodos membros (ou dados) são invocados sobre um ponteiro ou sobre uma referência

```
objprt->function();  
objref.function();
```

friend

- Todos os membros privados de uma classe são inacessíveis do exterior.
 - Se quisermos que uma função (**não membro**) tenha acesso aos membros privados de uma classe, podemos declarar esta função como amiga (**friend**).

friend

```
class my_class2;          // declaração forward
class my_class1 {
public:
    friend void compare(my_class1&, my_class2&);
    my_class1(int A) : a(A) {}
    // ...
private:
    int a;
};
class my_class2 {
public:
    friend void compare(my_class1&, my_class2&);
    my_class2(int A) : a(A) {}
    // ...
private:
    int a;
};
void compare(my_class1 &c1, my_class2 &c2) {
    if (c1.a==c2.a) cout << "equal\n";
    else cout << " not equal\n";
}
```

Palavra chave **const**

- Utilização

- constantes

- ```
const int PI=3.14159;
```

- ponteiros

- ```
char* const letra = &c1; // ponteiro constante  
const char* const letra = &c1; // obj e ponteiro
```

- argumentos de funções

- ```
void funcA(Person &p) { p.age = 32; }
```

- ```
void funcB(const Person &p) {  
    // p.age++; // Erro!  
    cout << p.age << endl;  
}
```

- métodos constantes

Métodos constantes

- Um método declarado como constante pode ser chamado para objectos constantes e para objectos não constantes.
- Um método não constante não pode ser chamado por objectos constantes.

```
class my_class {  
    int i;  
    const int max;  
public:  
    my_class (int ii, int m) : i(ii), max(m) {};  
    my_class inc () { if (i < max) i++; return *this; };  
    void Display () const { cout << i << endl; };  
};
```

```
my_class obj6(1, 50);  
const my_class obj7(1, 10);  
  
obj6.Display();  
obj7.Display();  
  
obj6.inc();  
obj7.inc(); //erro
```

Cada função que não modifica o estado do objecto tem que ser declarada como constante!!!

Métodos constantes

- Os construtores e destrutores nunca podem ser constantes
 - quase sempre modificam o estado do objecto.
- Quando uma função-membro constante é definida fora do corpo da classe, deve levar o sufixo **const** porque faz parte do tipo da função:

```
void my_class::Display () const
{
    cout << i << endl;
};
```

- Nas funções-membros constantes o ponteiro **this** é do tipo:

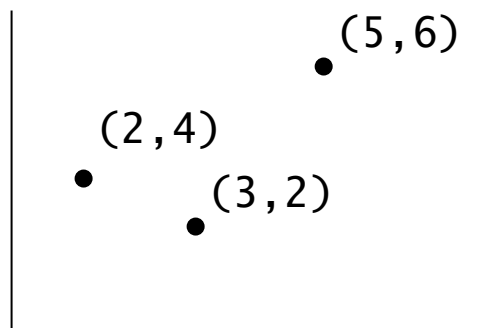
```
const my_class *const this;
```

Redefinição de operadores – exemplo “operador+”

- Consideremos a class Point.

```
class Point {  
    public :  
        // ...  
    private:  
        int x, y;  
};
```

- Para adicionarmos pontos precisamos de adicionar os valores de x e de y de cada ponto $(2,4) + (3,2) = (5,6)$



Solução 1

- Como função não membro o resultado será o seguinte:

```
Point operator+(const Point &lhs, const Point
    &rhs) {
    return Point(lhs.x + rhs.x, lhs.y + rhs.y);
}
int main() {
    Point a(3,7), b(18,2), r;
    r = a + b;
    // r = operator+(a,b)
    r.show();
    . . .
}
```

- Declaração na classe Point:

```
friend Point operator+(const Point &lhs, const
    Point &rhs);
```

Solução 2

- Como função membro o resultado será o seguinte:

```
class Point {
    public :
        Point operator+(const Point &rhs) const;
    // ...
    private:
        int x, y;
};

// Member operator function to add two Point's
Point Point::operator+(const Point &rhs) const
{
    return Point( x + rhs.getX(), y + rhs.getY());
}
```

Solução 2

- No main

```
int main() {  
    Point a(3,7), b(18,2), r;  
    r = a + b;  
    // r = operator+(a,b)  
    r.show();  
    . . .  
}
```

- Output (21,9)

Standard Template Library

STL is a general-purpose C++ library of generic algorithms and data structures.

1. Containers store *collections of objects*
 - `vector`, `list`, `deque`, `set`, `multiset`, `map`, `multimap`
2. Iterators *traverse containers*
 - random access, bidirectional, forward/backward ...
3. Function Objects encapsulate *functions as objects*
 - arithmetic, comparison, logical, and user-defined ...
4. Algorithms implement *generic procedures*
 - `search`, `count`, `copy`, `random_shuffle`, `sort`, ...
5. Adaptors provide an *alternative interface* to a component
 - `stack`, `queue`, `reverse_iterator`, ...

An STL Line Reverser

```
#include <iostream>
#include <stack>           // STL stacks
#include <string>          // Standard strings

void rev(void)
{
    typedef stack<string> IOStack; // instantiate the template
    IOStack ioStack;              // instantiate the template class
    string buf;

    while (getline(cin, buf)) {
        ioStack.push(buf);
    }
    while (ioStack.size() != 0) {
        cout << ioStack.top() << endl;
        ioStack.pop();
    }
}
```

Java Simplifications of C++

- no pointers — **just references**
- no functions — can declare **static** methods
- no global variables — use **public static** variables
- no destructors — **garbage collection** and **finalize**
- no linking — **dynamic class loading**
- no header files — can define **interface**
- no operator overloading — **only method overloading**
- no member initialization lists — call **super** constructor
- no preprocessor — **static final constants** and automatic inlining
- no multiple inheritance — **implement multiple interfaces**
- no structs, unions, enums — **typically not needed**