



CAP 365-3 Paradigmas e Ferramentas de Desenvolvimento de Software

Revisão de C++
Lúbia Vinhas
INPE-DPI

Linguagem C++ (Aula 1)



Quem é?!

- Bjarne Stroustrup
- <http://www.research.att.com/~bs/homepage.html>
- C++ foi criada e desenvolvida nos AT&T Bell Laboratories na década de 80
- C++: nome pretende dar a idéia de incremento em relação a linguagem C (++ é o operador de incremento)



Apresentando C++

- C++ descende diretamente de C e mantém a maior parte de C
- C++ é uma linguagem de propósito geral que:
 - é melhor que C
 - suporta abstração de dados
 - suporta programação orientada a objetos
 - suporta programação genérica
- Compiladores Free para C++
 - Linux: Free Software Foundation: G++ (www.fsf.org)
 - MS-Windows: Cygwin (<http://www.cygwin.com/>)



Como aprender C++?

- Cocentrando-se em conceitos e questões de design e não em detalhes técnicos da linguagem
- Conhecendo os paradigmas de programação:
 - Procedural: *decida quais procedimentos você quer; use os melhores algoritmos que encontrar*
 - Modular : *decida quais módulos você quer; particione o programa de modo a esconder os dados dentro dos módulos*
 - Abstração de Dados: *decida quais tipos você quer; providencie um conjunto completo de operações para cada tipo*
 - Orientada-a-objetos: *decida que classes você quer; forneça um conjunto completo de operações para cada classe; explicita as coisas comuns através de herança*
 - Programação genérica: *decida que algoritmos você quer; parametrize-os de forma que eles funcionem para diversos tipos e estruturas de dados desejados*
- Usando adequadamente as ferramentas da linguagem que permitem a implementação de cada paradigma



Bibliografia

- *C++ Programming Language Third Edition* - Bjarne Stroustrup
- *Effective C++: 50 Specific Ways to Improve Your Programs and Design* - Scott Meyers
- *More Effective C++: New Ways to Improve Your Programs and Designs* - Scott Meyers
- *C++ Templates The Complete Guide* - David Vandevoorde, Nicolai M Josuttis
- *The C++ Standard Library: A Tutorial and Reference* - Nicolai M Josuttis
- *Effective STL : 50 Specific Ways to Improve Your Use of the Standard Template Library* - Scott Meyers
- *Modern C++ Design, Generic Programming and Designs Patterns Applied* - Andrei Alexandrescu
- The C++ Report
- [C/C++ Users Journal](#)
- WWW



Tipos Fundamentais

- C++ oferece um conjunto de tipos que correspondem as unidades básicas de armazenamento mais comuns
- *Built-in*
 - Tipo booleano: `bool` (`true` ou `false`)
 - Tipo caracter: `char` (`'a'`, `'b'`, ..., `'z'`) (`unsigned`)
 - Tipo inteiro: `int` (`1`, `2`, ...) (`unsigned`) (`short`, `long`)
 - Tipos ponto-flutuantes: `double` (`1.2666`, `4.87`)
- *User Defined*
 - Enumeradores para criar conjuntos de valores específicos:

```
enum hemis { south=0; north=1};
```
 - Tipo para significar ausência de informação: `void`



Elementos básicos

- A partir dos tipos fundamentais podem ser construídos
 - Ponteiros: `int*`
 - Arrays: `char[]`
 - Referências: `double&`
 - Estruturas e Classes
- Todo identificador em C++ deve ser declarado antes de ser usado e deve possuir um tipo

C++ não exige que variáveis sejam declaradas no início do programa, portanto, retarde ao máximo sua declaração. Declare-as logo antes de onde serão usadas e não antes
- Tamanho dos tipos nativos são dependentes de implementação

O operador `sizeof` retorna o tamanho do tipo em múltiplos do tamanho de um `char` ou 1



Arrays

- Para um dado tipo T , $T[n]$ é um **vetor** de n elementos do tipo T
- Devem ser inicializados com tamanhos constantes:

```
int v1[10];
```

```
int v2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

- Nome de um array pode ser utilizado como ponteiro para o elemento inicial

```
int* p1 = v2;
```

```
// (*p1) é igual a 1
```

```
p1++;
```

```
// (*p1) é igual a 2
```


Elementos Básicos

- **Ponteiros:** para um dado tipo T, T* é do tipo ponteiro para T

```
char c = 'a';  
char* pc = &c;
```



```
int* pi;           // ponteiro para um int  
char** ppc;        // ponteiro para um ponteiro para um int  
int* ap[15];       // vetor de 15 ponteiros para int  
int (*fp)(char*);  // ponteiro para uma função com 1 char* como argumento  
int* f(char*);     // função com char* como argumento e retornando int*
```

- **Dereferenciamento** de um ponteiro retorna o objeto para o qual ele aponta

```
char c2 = *pc;
```



Elementos Básicos

- Declaração de variáveis

- `int x, y;`
- `int* x, y;`
- `int v[10], *pv;`

- Escopo

```
void f()
{
    int x;
    x = 1;
    {
        int x;
        x = 2;
    }
}
```

- Declaração e inicialização

- `int x = 1;`
- `int v[] = {1, 2};`

- **Typedef** : novo nome para um tipo

```
typedef unsigned int UINT;
```

```
UINT x, y;
```



Elementos básicos

- **Referências:** são nomes alternativos para objetos

```
int i=1;

int& r=i;           // r e i referenciam o mesmo inteiro

void incr(int& aa) { aa++; }

incr(i);           // i = 2;
```

- **Variáveis estáticas**

```
void f()
{
    static int n = 0;
    int x = 0;
    cout << "n = " << n++ << "x = " << x++;
}
```



Referências versus ponteiros

- Referências sempre se referem a um objeto existente enquanto que ponteiros podem ser nulos

```
void printDouble(const double& rd)
```

```
{
```

```
    cout << rd;
```

```
}
```

```
void printDouble(const double* pd)
```

```
{
```

```
    if (pd)
```

```
        cout << *pd;
```

```
}
```

Objeto não pode
ser modificado

// não é preciso testar

// verifica se ponteiro nulo

- Ponteiros podem mudar o objeto que apontam, referências não



void*

- Ponteiros para `void`

Qualquer ponteiro pode ser associado a ponteiro do tipo `void*` (incluindo outro `void*`). Dois `void*` podem ser comparados para igualdade. Um `void*` pode ser explicitamente convertido para outro ponteiro



Qualificador const

- Especifica uma restrição semântica

```
char* p = "Hello";           // ponteiro ã const, conteúdo ã const
const char* p = "Hello";     // ponteiro ã const, conteúdo const
char* const p = "Hello";     // ponteiro const, conteúdo ã const
const char* const p = "Hello"; // ponteiro const, conteúdo const
```

Se existir `const` à esquerda do `*`, o conteúdo é constante, se existir à direita o ponteiro é constante, se estiver a direita e a esquerda ponteiro e conteúdo são constantes

- Constantes `const m = 10;`

Podem ser usados como limitadores de arrays: `char v[m];`



Estruturas

- Um vetor agrega elementos do mesmo tipo. Uma estrutura agrega elementos de tipos arbitrários

```
struct Address {  
    char* name;  
    int    number;  
    int    street;  
    long   zip;  
    char*  town;  
    char   state[2];  
};
```

- Estruturas definem um novo tipo: `Address myAddress;`
- Elementos da estrutura são acessados com o operador *dot* (`.`): `myAddress.name`



Objetos na memória livre

- Ponteiros alocam objetos na área de memória livre, onde existem até que sejam explicitamente destruídos: `new`, `delete`, `new[]` e `delete[]`

```
{  
    Adress tempAddr;           // desaparece quando sai do escopo  
}  
  
Adress* myAddr = new Adress;   // objeto individual permanece  
delete myAddr ;               // até ser destruído  
  
int* q = new int[10];          // array de objetos  
delete []q;
```




Statements

```
try { statement-list} catch (expression) {statement-list}
```

```
if (condition) statement
```

```
if (condition) statement else statement
```

```
switch (condition ) statement
```

```
while (condition) statement
```

```
do statement while (expression)
```

```
for (init; condition; expression) statement
```

```
case constante-expression : statement
```

```
default : statement;
```

```
break;
```

```
continue;
```

```
return expression;
```



Funções

- Funções

```
void swap(int*, int*);
```

```
// declaração
```

```
void swap (int* p, int* q)
```

```
// definição
```

```
{
```

```
    int t = *q;
```

```
    *p = *q;
```

```
    *q = t;
```

```
}
```



Funções

- Passagem de parâmetros
 - Referências e ponteiros permitem a alteração dos parâmetros
 - Parâmetros simples são passagens por cópia

```
void f(int val, int& ref)
```

```
{
```

```
    val++;
```

```
    ref++;
```

```
}
```

```
int
```

```
main()
```

```
{
```

```
    int i=1;
```

```
    int j=1;
```

```
    f(i,j);
```

```
    return 0;
```

```
}
```

```
i = 1 e j = 2
```



Funções

- Forma padrão: passar parâmetros por referência

- Evita cópias desnecessárias
- Se necessário passe como referência constante

```
void h (Large& arg) {...}
```

```
void g (const Large& arg) {...}
```

- Arrays são passados como argumentos através de ponteiro para sua primeira posição
- Argumento default

```
void print(int value, int base=10);
```

```
print(5) ou print(4,2);
```

Argumentos com valor default estão sempre no final da lista de parâmetros



Funções

- Valor de retorno:

```
int f() { int i=0; return i; }           // OK: cópia
```

```
int* fp() { int local=0; return &local; } // erro!
```

```
int& fr() { int local=0; return local; }  // erro!
```

Objetos com escopo interno a função não podem ser retornados por referências ou ponteiros

- Sobrecarga de funções: mesmo nome mas diferentes parâmetros

```
void print(int);
```

```
void print(const char*);
```

```
void print(double);
```

Não existe sobrecarga por diferença de valor de retorno, somente por diferença de parâmetros



Namespaces

- Um namespace é um mecanismo usado para expressar um agrupamento lógico de classes, objetos ou funções globais sob um determinado nome

```
namespace LibX {  
    double f1() { /*...*/ }  
    double h() { /*...*/ }  
    int a;  
}  
  
namespace LibY {  
    double g1() { /*...*/ }  
    double h() { /*...*/ }  
    int a;  
}
```

- Especialmente útil para evitar conflitos de nomes, para isso é usado o qualificador :: precedido pelo nome do namespace
LibX::a é diferente de LibY::a



Namespaces

- Interfaces podem estar separadas da implementação, desde que qualificador seja usado na implementação

```
namespace LibX
{
    double f1();
    double h();
    int a;
}
double LibX::f1(bool get) { /*...*/ }
double LibX::h(bool get) { /*...*/ }
```

- Namespaces podem ser acrescentados a qualquer momento

```
namespace LibX
{
    int b;
}
```



Diretiva *using*

- É usada para evitar a repetição da qualificação de nomes

```
using namespace LibX; // disponibiliza todos os os elementos da LibX
using LibY::gl;       // disponibiliza gl da LibY

main()
{
    bool b = false;
    fl(b);             // LibX::fl
    gl(b);              // LibY::gl
}
```




Tratamento de Erros

- Supor os módulos:

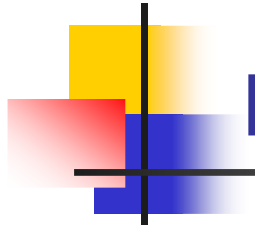
```
int ReadFile(char* filename)
{
    // open file
    // le o arquivo
    // retorna o valor lido
}
```

Biblioteca

```
...
int x = ReadFile("teste.txt");
```

Aplicação

- O que acontece se o arquivo não pode ser aberto?



Exceções

- Quando um programa é composto de vários módulos separados, especialmente quando esses módulos estão em bibliotecas separadas, o tratamento de erros deve contemplar dois aspectos:
 1. O apontamento de erros que não podem ser resolvidos localmente
 2. O tratamento de erros que são detectados em outro lugar
- O autor da biblioteca pode detectar erros em tempo de execução, mas não tem idéia do que fazer com eles
- O autor da aplicação sabe como tratar os erros mas não é capaz de detectá-los
- O uso de *exceções* é o mecanismo de C++ para separar apontamento/detecção do erro do seu tratamento



Exceções

- O mecanismo de *Exception* ajuda no apontamento de erros:

```
struct Range_error {  
    int i;  
    Range_error (int ii) { i=ii; }  
}
```

```
char to_char(int i)  
{  
    if (i < 0 || i > 255)  
        throw Range_error(i);  
    return i;  
}
```

- A função `to_char` ou retorna o caracter correspondente a `i` ou “joga” um erro de intervalo (`Range_error`)



Throw, Try e Catch

- Uma função que sabe o que fazer no caso de um erro de intervalo indica que é capaz de “recolher” a exceção e tomar uma decisão

```
void g(int i)
{
    try {
        char c = to_char(i);
        //...
    }
    catch(Range_error)
    {
        cout << “Erro de limite ”;
    }
}
```



Throw, Try e Catch

- Uma função que sabe o que fazer no caso de um erro de intervalo indica que é capaz de “recolher” a exceção e tomar uma decisão

```
void g(int i)
{
    try {
        char c = to_char(i);
        //...
    }
    catch(Range_error& err)
    {
        cout << "Erro de limite " << err.i;
    }
}
```

Usa a exceção
retornada

<< err.i;



Catching Exceções

- Se em qualquer parte do bloco *try* alguma exceção é jogada, o *catch* será examinado
- Se a exceção jogada for do tipo esperado pelo *catch* esse será executado
- Se não for do tipo esperado, o *catch* será ignorado
- Se uma exceção é jogada e não existe nenhum *try-block* o programa termina a execução
- Basicamente o tratamento de exceções é uma forma de transferir o controle para uma código designado por uma função
- Quando necessário, informações sobre o erro podem ser passadas
- O mecanismo de tratamento de erros por exceção é fornecido com o objetivo de tratar erros síncronos

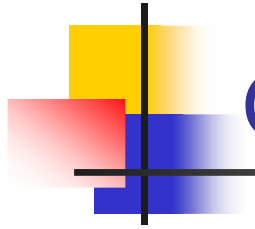


Diferentes tipos de exceção

- Tipicamente é de se esperar que possam ocorrer diferentes tipos de erro de execução. Cada tipo de erro pode ser mapeado para um exceção com nome diferente

```
struct zero_divide{};
struct sintax_error {};

try{
    expr(false);
    // esse ponto é atingido se expr não joga nenhuma exceção
}
catch (sintax_error)
{ // trata erro de sintaxe }
catch (zero_divide)
{ // trata divisão por zero }
// esse ponto é atingido se expr não causou exceção ou se os handlers trataram
a exceção mas não jogaram outra exceção para o nível de cima e também não
fizeram return
```



Classes

- Mecanismo de C++ que permite aos usuários a construção de seus próprios tipos (*user defined*) , que podem ser usados como tipos básicos (*built-in*)
- Um tipo é a representação de um conceito. Exemplo: tipo `float` e as operações `+`, `-`, `*`, `/`, formam a representação do conceito matemático de um número real
- Uma classe é um tipo definido pelo usuário, que não tem similar entre os tipos nativos. Possui atributos (ou membros) e métodos



Classes

```
class Date{  
private:  
    int d, m, y; // representacao  
  
public:  
    void init(int dd, int mm,  
              int yy);  
    void year(int n);  
    void month(int n);  
    void day(int n);  
    int dayOfYear();  
};
```

Interface

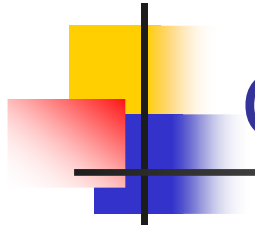
```
void  
Date::init(int dd, int mm, int yy)  
{...}  
void  
Date:: year(int n);  
{...}  
void  
Date:: month(int n);  
{...}
```

Implementação

Objeto da classe

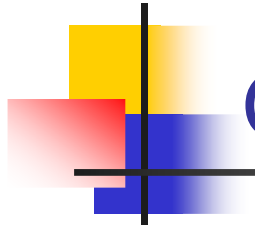
```
Date lubia_nasc;  
lubia_nasc.init(1,3,1969);  
  
Date natalia_nasc;  
natalia_nasc.init(1,5,2003)
```

Cliente



Classes

- Classes refletem dois dos princípios fundamentais de orientação-a-objetos: abstração de dados e encapsulamento
- **Abstração:** podem ser criados tipos abstratos, particulares de uma aplicação que se comportam como tipos nativos
- **Encapsulamento:** desde que não se altere a parte pública da interface, a parte de representação e a implementação dos métodos podem ser alterados sem que o cliente tenha que ser modificado
- Use as ferramentas da linguagem e os princípios de orientação a objetos de forma que sua classe seja fácil de usar corretamente e difícil de usar incorretamente
- Ler o artigo: “The Most Important Design Guideline? - Scott Meyers - IEEE Software Julho/Agosto de 2004 “



Controle de acesso

- Os modificadores de controle de acesso são aplicados tanto a membros como a métodos

Parte `private` : é acessível somente pelos outros membros da mesma classe ou por classes `friend`

Parte `protected`: é acessível também pelas classes derivadas de uma classe

Parte `public`: é acessível a partir de qualquer ponto onde a classe seja visível



Construtores

- São a forma de inicialização de objetos de uma determinada classe

```
class Date{  
    //...  
    Date(int, int, int); // construtor  
};
```
- Construtores podem ser sobrecarregados, ou seja, podem ser fornecidas diferentes versões com diferentes tipos de parâmetros
- Se não for fornecido nenhum construtor, o compilador irá criar um *default*
- ```
Date hoje = Date(2,10,2002);
Date natal(25,12,2004);
Date aniver; // Erro! Não existe Date()
Date versao_3(10,10); // Erro! Não existe Date com 3 args
```



# Construtores

---

- Construtor Default não recebe parâmetros (`Date()`)  
`Date d1;`
- Aceitam valores default  
`Date::Date(int, int, int yy=2004)`
- Construtor de cópia: constrói um objeto novo a partir de um outro já criado  
`Date d2(d1);`  
`Date d3 = d1;`
- Passar parâmetros por valor, implica chamada no construtor de cópia, portanto prefira a passagem de parâmetros por referência

```
void Function(Date d2); // chama construtor de cópia
void Function(Date& d2); // não chama construtor de cópia
void Function(const Date& d2); // protege parâmetro
```



# Construtores

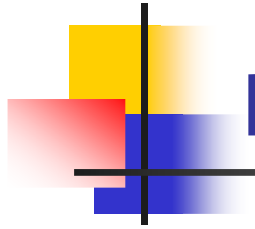
---

- Lista de inicialização é preferível ao invés de atribuição no corpo do construtor:

```
Date::Date(int dd, int mm, int yy):
 d(dd), m(mm), y(yy) { }
```

```
Date:: Date(int dd, int mm, int yy)
{ d = dd; m = mm; y = yy; }
```

- Defina a lista de inicialização na mesma ordem com que os membros foram declarados
- Certifique-se que todos os membros da classe foram inicializados principalmente ponteiros



# Destrutores

---

- São chamados cada vez que um objeto de uma classe sai fora de escopo ou é explicitamente destruído

- Devem liberar toda a memória que foi alocada no construtores ou em algum método da classe

```
String::String(int n) {data = new char[n]; }
```

```
String::~~String() { delete [] data; }
```

- Caso não sejam implementados, o compilador irá fornecer um



# Métodos

---

- Métodos podem ser declarados constantes, ou seja, garantem que não alteram membros da classe

```
class Date{
 //...
 int day() const {return d;}
 void incDay() { d=d+1; }
};
```

- Métodos constantes podem ser chamados por objetos constantes e não constantes. Métodos não constantes não podem ser chamados por objetos constantes

```
const Date cd(1,12,2004);
Date ncd(10,11,2001);
int d = cd.day(); // ok
int d2 = ncd.day(); // ok
ncd.incDay(); // ok
cd.incDay(); // error!
```





# Membros e Métodos estáticos

---

- Pertencem a classe e não a cada objeto da classe

```
class Date{
private:
 int d, m, y;
 static Date default_date_;
public:
 Date (int dd, int mm, int yy);
 //...
 static void set_default(int, int, int);
};
```

- Membros estáticos devem ser definidos em algum lugar

```
Date::default_date(16,12,1770);
```

- Métodos estáticos não precisam de um objeto para serem chamados

```
Date::set_default(4,5,1945); // estático
Date d(1,2,2004);
d.dayOfYear(); // não estático
```



# Sobrecarga de operadores

---

- Além de métodos, classes podem redefinir operadores como se fossem seus métodos. Dessa forma podem ser manipuladas em uma notação conveniente, similar aos tipos nativos

```
class Complex {
 double re, im;
public:
 Complex(double r, double i): re(r), im(i) {}
 Complex operator+(Complex rhs);
};
```

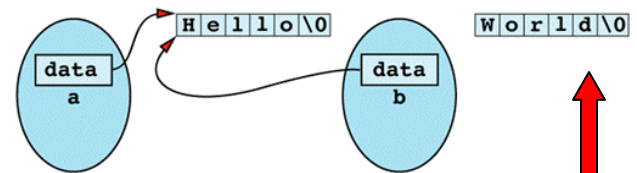
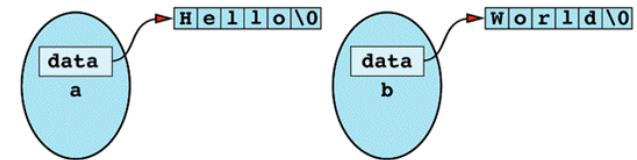
```
Complex a = Complex(1.3,1.8);
Complex b = Complex(1.3,2.0);
Complex c = a + b; // equivale a "c = a.operator+(b)"
 // notação equivalente ao op + sobre inteiros
```

# Cópia de Objetos

- Por default objetos podem ser copiados. A cópia é feita copiando-se membro a membro tanto na inicialização quanto na associação  

```
Date d1(2,2,2002), d2;
d2 = d1;
```
- Forneça suas próprias versões do construtor de cópia e operador de associação se você tem ponteiros em suas classes para prevenir *leaks* de memória

```
class String {
private:
 char* data;
public:
 String(char* s) {...}
};
String a("Hello"), b("World");
b = a;
```



Leak de memória

# Operator =

- Garanta que o valor de retorno seja uma referência a this

```
Date& operator=(const Date rhs)
```

```
{...; return *this;}
```

```
Date d1, d2, d3;
```

```
d1 = d2 = d3; // torna possível o encadeamento
```

- Verifique a auto referência

```
String& String::operator=(const String& rhs)
```

```
{
```

```
 delete [] data; // invalida data
```

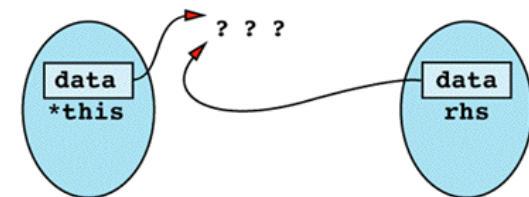
```
 data = new char[strlen(rhs.data) + 1];
```

```
 strcpy(data, rhs.data);
```

```
 return *this;
```

```
}
```

```
String a; a=a;
```





# Operator =

---

- Garanta que o valor de retorno seja uma referência a `this`

```
Date& operator=(const Date rhs)
 {...; return *this;}
Date d1, d2, d3;
d1 = d2 = d3; // torna possível o encadeamento
```

- Verifique a auto referência

```
String& String::operator=(const String& rhs)
{
 if (rhs == *this)
 return *this;
 delete [] data;
 data = new char[strlen(rhs.data) + 1];
 strcpy(data, rhs.data);
 return *this;
}
String a; a=a;
```

- Garanta que todos os membros da classe sejam associados



# Forma canônica

---

```
class Exemp {
public:
 Exemp(); // construtor
 virtual ~Exemp(); // virtual se classe base
protected:
 // Dados que meus filhos precisam
private:
 // Meus dados pessoais
 Exemp (const Exemp& rhs) {} // construtor de cópia
 Exemp& operator= (const Exemp& rhs) {} // atribuição
}
```



# Funções amigas

---

- A declaração normal de uma função em uma classe garante logicamente que:
  1. O método pode acessar as partes privadas da classe onde é declarado
  2. O método existe no escopo da classe
  3. O método deve ser invocado através de um objeto da classe

- As funções estáticas eliminam a restrição 3

```
Date::set_default(1,1,2002);
```

- As funções `friends` eliminam as restrições 3 e 2

```
class Date { ...
 int d, m, y;
public:
 friend Date tomorrow(const Date& today);
}

Date tomorrow(const Date& today)
{ Date res(today.d+1, today.m, today.y); return res; }
```



# Classes amigas

---

- Permitem que uma classe acesse as partes privadas de outra

```
class CSquare;
class CRectangle {
 int width, height;
public:
 void convert (CSquare a);
};

class CSquare {
private:
 int side;
public:
 friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
 width = a.side;
 height = a.side;
}
```





# Referências

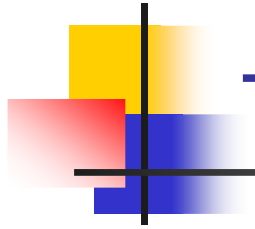
---

- Passagem de parâmetros e retorno por referência constante é mais eficiente e seguro

```
Person returnPerson(Person p) { return p; }
```

```
Person& returnPerson(const Person& p) { return p; }
```

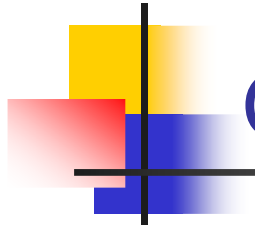
- No entanto, quando o valor de retorno é criado na função, não se pode retornar referência



# Tipos Concretos

---

- As classes mostradas até agora são tipos concretos, ou seja, são definidos pelo usuário de forma a serem operadas como tipos nativos
- Tipos definidos pelo usuário eficientes:
  - Ler o capítulo 10 do C++ 3rd Edition a partir do item 10.3
- Sobrecarga de Operadores
  - Ler o capítulo 11 do C++ 3rd Edition



# Classes

---

- Mecanismo de C++ que permite aos usuários a construção de seus próprios tipos (*user defined*) , que podem ser usados como tipos nativos (*built-in*)
- Um tipo é a representação de um conceito. Exemplo: tipo `float` e as operações `+`, `-`, `*`, `/`, formam a representação do conceito matemático de um número real
- Uma classe é um tipo definido pelo usuário, que não tem similar entre os tipos nativos. Possui atributos (ou membros) e métodos



# Classes

```
class Date{
private:
 int d, m, y; // representacao

public:
 void init(int dd, int mm,
 int yy);
 void year(int n);
 void month(int n);
 void day(int n);
 int dayOfYear();
};
```

Interface

```
void
Date::init(int dd, int mm, int yy)
{...}
void
Date:: year(int n);
{...}
void
Date:: month(int n);
{...}
```

Implementação

Objeto da classe

```
Date lubia_nasc;
lubia_nasc.init(1,3,1969);

Date natalia_nasc;
natalia_nasc.init(1,5,2003)
```

Cliente



# Herança

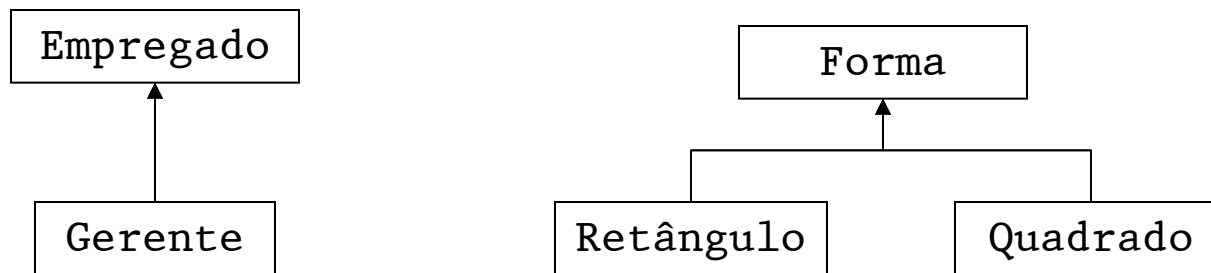
---

- Classes representam um tipo definido pelo usuário que se comporta como um tipo nativo
- Classes podem ser usadas para representar conceitos no domínio de aplicação
- Conceitos não existem isoladamente. Sua força está em seu relacionamento com outros conceitos
- Ex: Triângulos e Círculos relacionam-se entre si já que ambos são formas geométricas. Logo, pode-se explicitamente definir que uma classe Circle e uma classe Triangle são ambas derivadas de uma classe base Shape



# Herança

---



- Uma classe derivada de uma classe base herda as propriedades da classe base acrescentando ou especializando propriedades particulares
- Uma classe derivada é uma classe base, ou é um tipo de classe base
- Uma classe derivada sempre deve poder ser usada onde se espera uma classe base
- Como C++ indica derivação:  

```
class Empregado { ... };
class Gerente : public Empregado { ... };
```




# Herança

---

- Classes derivadas publicamente, possuem acesso a todos os membros públicos e protegidos das classes bases

```
class Empregado {
protected:
 string nome, sobrenome;
public:
 string nomeCompleto() const
 { return nome + ' ' +
 sobrenome; }
};
```



```
class Gerente : public Empregado {
 int departamento;
public:
 void print() const
 {
 cout << "meu nome e: " << nome;
 cout << "meu nome completo e: ";
 cout << nomeCompleto();
 }
};
```



# Herança

---

- Podem acessar explicitamente métodos das classes base

```
void Gerente::print() const
{
 Empregado::print();
 cout << "gerente";
}
```

- Construtores de classes derivadas não podem instanciar membros das classes base, mas podem chamar seu construtor

```
Empregado::Empregado(const string& n, const string& s):
 nome(n), sobrenome(s) {...}
```

```
Gerente::Gerente(const string& n, const string& s, int d):
 Empregado(n,s),
 departamento(d) {...}
```





# Herança (Cópias)

---

- Cópias de classes derivadas para classes básicas podem sofrer problema de *slicing*

```
void f1(const Gerente& g)
{
 Empregado e = g; // apenas a parte Empregado de g é copiada
 e = g;
}
void f2 (Empregado e) // parâmetro passado por valor => slicing
{
 e.print();
}
main()
{
 Gerente g;
 f2(g); // executa print de Empregado e não de Gerente
}
```



# Herança (Cópias)

---

- Cópias de classes derivadas para classes básicas podem sofrer problema de *slicing*

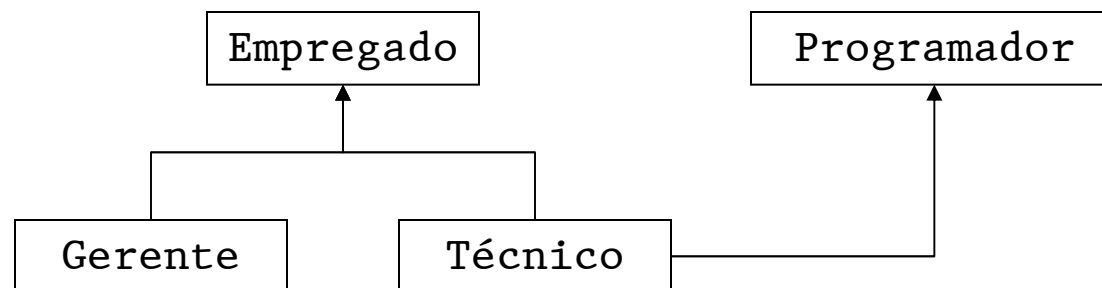
```
void f1(const Gerente& g)
{
 Empregado e = g; // apenas a parte Empregado de g é copiada
 e = g;
}
void f2 (const Empregado& e) // o.k.
{
 e.print();
}
main()
{
 Gerente g;
 f2(g); // executa print de Empregado e não de Gerente
}
```

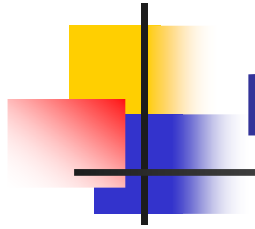


# Herança

---

- Objetos são construídos na seguinte ordem: classe base, seus membros e a classe derivada
- Objetos são destruídos na ordem inversa: a classe derivada, seus membros e a classe derivada
- Construtores e operadores de associação não são herdados
- Podem ser derivadas mais que uma classe da mesma base
- Podem ser criadas classes derivadas de mais que uma classe base





# Polimorfismo

---

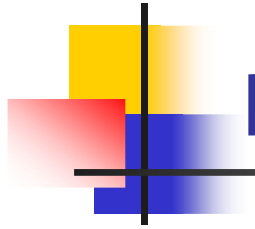
- Objetos de classes derivadas podem ser tratados como objetos de classes base quando manipulados através de ponteiros e referências

```
void Promove(Empregado& emp)
{...}

void Promove2(Empregado* emp)
{...}
```

```
Empregado e1("Luis","Azevedo");
Gerente g1("Antonio","Camargo",1);
```

```
Promovel(e1); Promove2(&e1);
Promovel(g1); Promove2(&g1);
```



# Polimorfismo

---

- Eventualmente, é necessário fornecer comportamentos diferentes entre diferentes classes derivadas ou entre uma classe base e uma classe derivada

```
class Empregado
{
 void bonus(); // Gerentes ganham 15%
 // Técnicos ganham 10%
}

void ProcessaPromocao(Empregado& emp)
{
 emp.bonus();
}
```

- Como garantir que uma função se comporte diferente caso seja aplicado a um objeto de uma classe derivada ou outra?



# Polimorfismo

---

- Permitem a reimplementação de funções definidas na classe base pelas classes derivadas
- Compilador irá garantir a correspondência entre o objeto e a função correta

```
classe Empregado {
 double salario;
public:
 virtual void bonus(){};
};

class Gerente : public Empregado {
public:
 void bonus()
 { salario = salario + 0.15*salario; }
};

class Tecnico : public Empregado {
public:
 void bonus()
 { salario = salario + 0.10*salario; }
};
```

```
void ProcessaPromocao(Empregado& emp)
{
 emp.bonus();
}

Tecnico tecnico1;
Gerente gerentel;

ProcessaPromocao(gerentel);
ProcessaPromocao(tecnico1);
```



# Herança Pública

- Se B é uma base *public* seus membros públicos podem ser usados por quaisquer funções. Seus membros *protected* podem ser usados por membros e *friends* de D e seus derivados. Qualquer função pode converter D\* para B

```
class Base {
 int bPrivateMember_;
 void bPrivateFunction()
 { bPrivateMember_ = 2; }

protected:
 int bProtectedMember_;
 void bProtectedFunction()
 { bProtectedMember_ = 1; }

public:
 int bPublicMember_;
 void bPublicFunction()
 { bPublicMember_ = 0; }
};
```

```
class Derived : public Base {
public:
 void test(){
 bPublicMember_ = 1;
 bProtectedMember_ = 1;
 bPublicFunction();
 bProtectedFunction();
 }
};
```

```
void f(Base* b) {...}

int main()
{
 Derived* g = new Derived();
 f(g);
}
```



# Herança Protegida

- Se B é uma base *protected*, seus membros públicos e privados só podem ser usados por funções e *friends* de D e por funções membros e *friends* das classes derivadas de D. Somente *friends*, membros e derivados de D podem converter D\* para B\*

```
class Base {
 int bPrivateMember_
 void bPrivateFunction()
 { bPrivateMember_ = 2; }

protected:
 int bProtectedMember_
 void bProtectedFunction()
 { bProtectedMember_ = 1; }

public:
 int bPublicMember_
 void bPublicFunction()
 { bPublicMember_ = 0; }
};
```

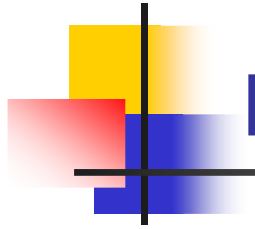
```
class Derived : protected Base {
public:
 void test(){
 bPublicMember_ = 1;
 bProtectedMember_ = 1;
 bPublicFunction();
 bProtectedFunction();
 }
};
```

```
void f(Base* b) {...}

int main()
{
 Derived* g = new Derived();
 f(g);
}
```

Conversão  
inacessível





# Herança Privada

---

- Se B é uma base *private* seus membros públicos e protegidos só podem ser usados por membros e *friends* de D. Somente *friends* e membros de D podem converter  $D^*$  para  $B^*$ . Herança privada é somente uma técnica de implementação, deve ser entendida como “implementa-através-de”



# Arrays e polimorfismo

- Arrays não devem ser tratados polimorficamente

```
class A {
public:
 A () { ... }
private:
 int member1;
};
```

```
class B: public A {
public:
 B () {...}
private:
 int memberB;
};
```

```
void f(const A array[], int n)
{
 for (int i=0; i<n; ++i)
 cout << array[i];
}
```

```
A A_array[10]; // um vetor de 10 A's
f(A_array,10); // o.k.

B B_array[10];
f(B_array,10); // Erro!
```

- Aliás: `A A_array[10]` só é possível se `A` possui construtor default (`A()`)



# Referências x Ponteiros

---

- Referência sempre se refere a um objeto que existe

```
void printDouble(const Empregado& emp)
{
 cout << emp.nome(); // não há necessidade de testar se rd existe
}
```

- Ponteiros podem apontar para o “nada”

```
void printDouble(const Empregado* emp)
{
 if (emp) { // checa se ponteiro é nulo
 cout << emp->nome(); }
}
```

- Ponteiros podem ser reassociados, referências não

```
string s1("Bom"), s2("Dia");
string& rs = s1; // rs refere-se a s1
string *ps = &s1; // ps aponta para s1
rs = s2; // rs ainda refere-se a s1,
 // mas seu valor agora é "Dia"
ps = &s2; // ps agora aponta para s2
```



# Destrutores Virtuais

- Destrutores de classes base devem ser declarados virtuais, para garantir que o destrutor da classe derivada seja chamado, quando essa seja manipulada através de um ponteiro para classe base

```
class A {
public:
 A () {++numA;}
 ~A() {--numA;}
private:
 static int numA;
};
```

```
class B: public A {
public:
 B () {++numB;}
 ~B() {--numB;}
private:
 static int numB;
};
```

```
void f(A *ptr)
{ delete ptr; }
```

```
A* x = new B();
f(x);
```

// destrutor de B nunca é chamado!



# Destrutores Virtuais

- Destrutores de classes base devem ser declarados virtuais, para garantir que o destrutor da classe derivada seja chamado, quando essa seja manipulada através de um ponteiro para classe base

```
class A {
 public:
 A () {++numA;}
 virtual ~A() {--numA;}
 private:
 static int numA;
};
```

```
class B: public A {
 public:
 B () {++numB;}
 ~B() {--numB;}
 private:
 static int numB;
};
```

```
void f(A *ptr)
{ delete ptr; }
```

```
A* x = new B();
f(x); // o.k.
```



# Classes Abstratas

---

- Servem para representar conceitos para os quais objetos concretos não existem
- Exemplo: *shape* faz sentido apenas como uma classe base para derivação de classes derivadas como *circle* ou *polygon*

```
class Shape{
 public:
 void rotate(int) { ????? }
 void draw() { ??? }
};
```

- Um objeto genérico *shape* não sabe se desenhar ou se rotacionar, portanto não pode ser criado



# Classes Abstratas

---

- Servem para representar conceitos para os quais objetos concretos não existem, mas podem ser definidos por um conjunto de métodos
- Exemplo: *shape* faz sentido apenas como uma classe base para derivação de classes derivadas como *circle* ou *polygon*

```
class Shape{
 public:
 void rotate(int) { error(1); }
 void draw() {error(2); }
};
Shape shp; // Legal mas não faz sentido
```

- Não é uma boa solução pois todas as operações sobre *shape* resultam em erros



# Classes abstratas

---

- Classes abstratas podem definir um ou mais métodos como **puramente virtuais** e não podem ser instanciadas

```
class Shape{
 public:
 virtual void rotate(int) = 0; ←
 virtual draw() = 0; ←
};
```

```
Shape s; // erro: variável de classe abstrata não é permitido
```





# Classes Concretas

---

- Uma classe derivada que implementa todas as funções puramente virtuais de uma classe base torna-se uma classe concreta

```
class Point { /*...*/};
```

```
class Circle : public Shape {
public:
```

```
 void rotate(int) { /*...*/}
```



```
 void draw() { /*...*/}
```



```
private:
```

```
 Point center;
```

```
 int radius;
```

```
};
```

```
Shape* c = new Circle();
```

```
C->rotate(45);
```



# Classes Concretas

---

- Uma classe derivada que não implementa as funções puramente virtuais da classe base mantém-se uma classe abstrata

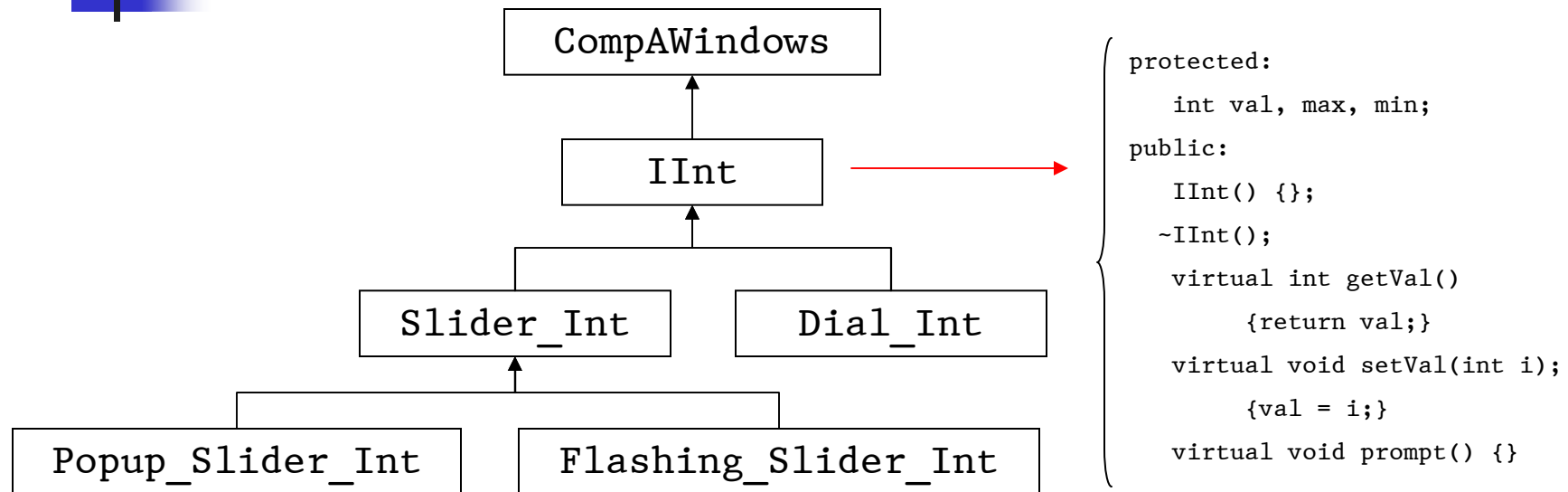
```
class Polygon : public Shape {
public:
 bool isClosed() { return true; }
};
```

```
Polygon b; // Erro: declaração de objeto de classe abstrata
```

```
class FourSides : public Polygon {
 Point ll;
 Point ur;
public:
 void rotate(int);
 void draw();
}
```

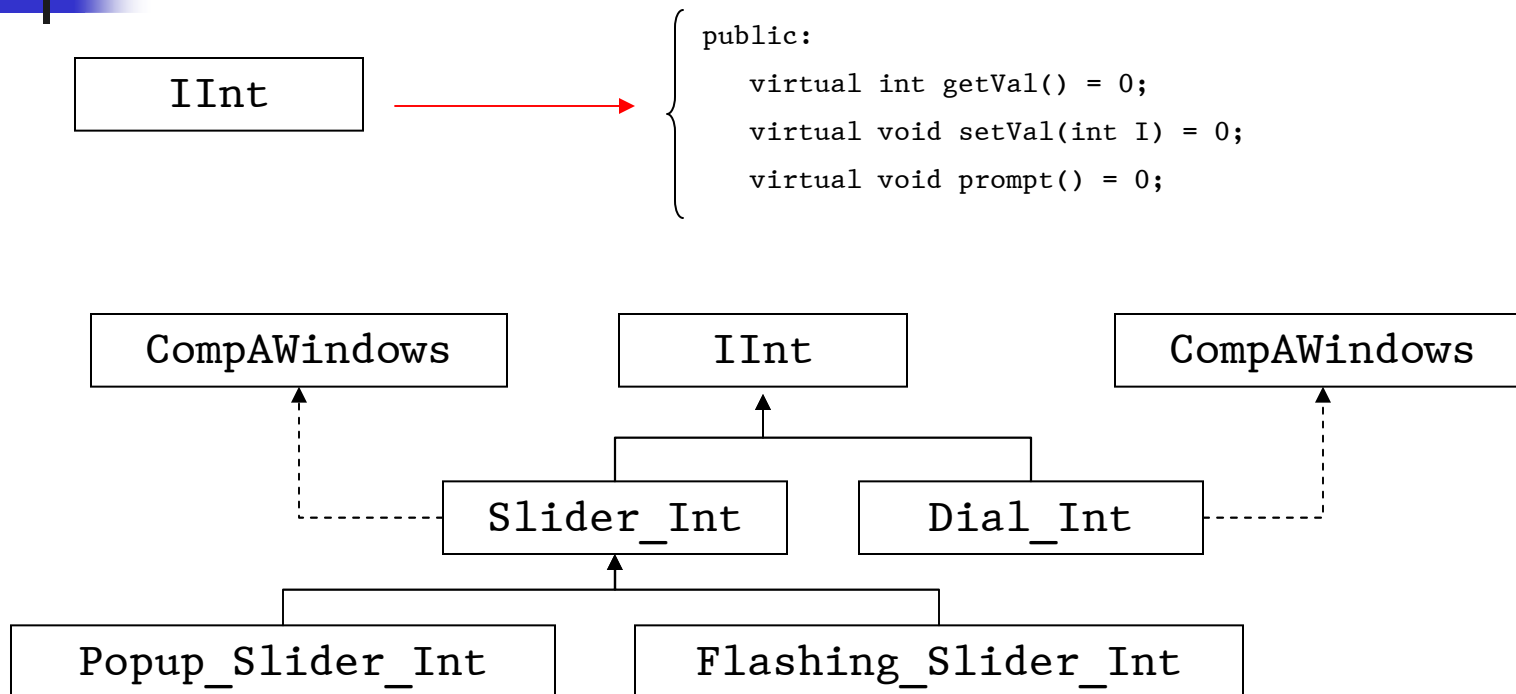
```
FourSides square; // OK!
FourSides rectangle;
```

# Projeto de Hierarquia de Classes



- Nem todas as classes derivadas precisam dos membros protegidos da classe base
- Derivar IInt de CompAWindows : não é intuitivo, é detalhe de implementação
- E se você quiser alterar o tool kit de GUI para companhia B?
- Qualquer mudança no toolkit implica na recompilação da sua aplicação

# Alternativa

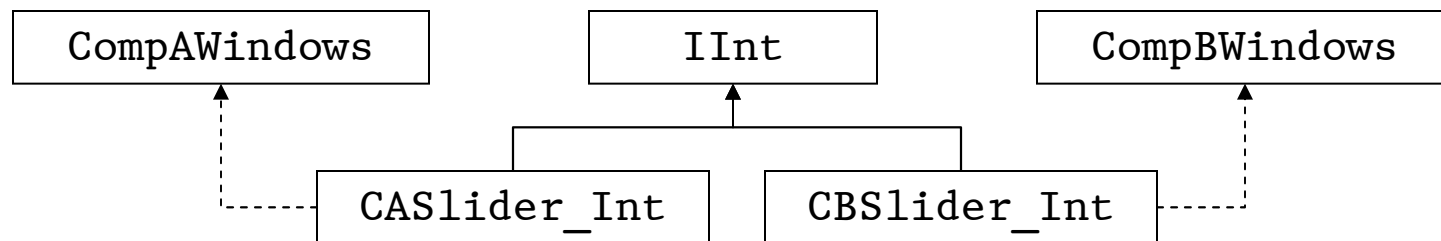


- IInt é uma interface abstrata
- A herança pública de IInt faz com que Slider\_Int implemente a interface
- A herança protegida de CompAWindows fornece os meios para essa implementação

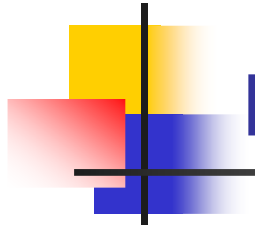


## Alternativa

---



- Permite a co-existência de diferentes implementações relacionadas a diferentes toolkits



# Hierarquia de classes

---

- Classes abstratas são **interfaces**
- Hierarquia é uma forma de construir classes incrementalmente
- Hierarquia clássica: classe base fornecem algumas funcionalidades úteis para clientes e também serve como *building blocks* para implementações mais especializadas ou avançadas
  - Grande suporte para a construção de novas classes desde que essas sejam altamente relacionadas as classes bases
  - Estão acopladas a implementação
- Hierarquia de classes abstratas representam conceitos sem expor detalhes de implementação
- Chamadas a métodos virtuais não custam mais que chamadas a quaisquer outros membros



# Questões de Design

---

- Classes relevantes para design:
  - Representar um conceito relevante
    - Domínio da aplicação
    - Artefatos de implementação
  - Expor uma boa interface
- A Classe ideal:
  - Tem a menor e mais bem definida dependência do resto do mundo
  - Tem uma interface que expõe o mínimo de informação para o resto do mundo



# Inclusão

---

- Quando uma classe precisa conter objetos de outra

```
class X {
 Y obj1;
 Y* obj2;
 setObj2(Y* y) { obj2 = y; };
}
```

- Se o valor do ponteiro não muda, as duas alternativas são equivalentes. Conter o objeto (e não o ponteiro) é mais eficiente
- Conter o ponteiro é útil quando seu valor muda ou quando seu valor deva ser passado por parâmetro





# Inclusão ou Derivação?

---

```
class B { /*...*/ }
class D: public B
{ /*...*/ }
```

*is-a*

```
class D {
public:
 B b;
};
```

*has-a*

- Faz parte da semântica do conceito permitir que D seja convertido para B?
- Inclusão permite type checking em tempo de compilação
- Derivação adia decisões até tempo de execução
- Prefere inclusão



# C++ conversores de tipos

---

- C++ provê novos operadores de conversão de tipos (*casting*):
- static\_cast: fornece uma maneira segura e portátil de converter tipos

```
double result = static_cast<double>(n1/n2);
void* p;
int* i = static_cast<int*> (p);
```

- const\_cast: remove ou adiciona o qualificador de constante

```
void update(MyClass* obj);
const MyClass myObj;
update(myObj); // erro: não é possível passar um const
 // para uma função que espera non const
update(const_cast<MyClass*> (myObj)); // o.k.
```



# C++ conversores de tipos

---

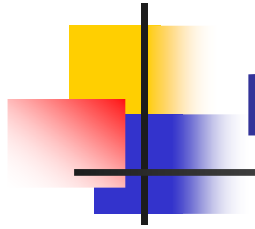
- reinterpret\_cast: converte tipos não relacionados, por exemplo ponteiros

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

- dynamic\_cast: converte um ponteiro de classe base para um ponteiro de classe derivada. Verifica-se a operação é válida, ou seja, serve para testar qual instancia de classe derivada está sendo tratada polimorficamente. Só pode ser aplicado a ponteiros e referências

```
class Base { /* ... */ };
class Derived : public Base {};
Base* b1 = new Derived;
Base* b2 = new Base;
Derived* d1 = dynamic_cast<Derived*>(b1); // sucesso
Derived* d2 = dynamic_cast<Derived*>(b2); // falha: retorna nulo
```

É necessário habilitar o compilador para permitir Run Time Information



# Projeto de Classes

---

- Objetivo: construir interfaces que são **mínimas e completas**
- Interface completa: permite que o cliente faça tudo que é razoavelmente necessário
- Interface mínima: contém o mínimo de métodos possível
  - 10 métodos : tratável
  - 100 métodos : difícil manutenção, afasta os clientes
- Não há uma receita de bolo. Critérios que justificam a inclusão de métodos a uma classe:
  - Se uma tarefa é implementada mais eficiente se for um método da classe
  - Se um método torna a classe substancialmente mais fácil de usar
  - Se um método irá prevenir erros por parte do cliente da classe
- Ler capítulo 12 do C++ 3rd Edition