



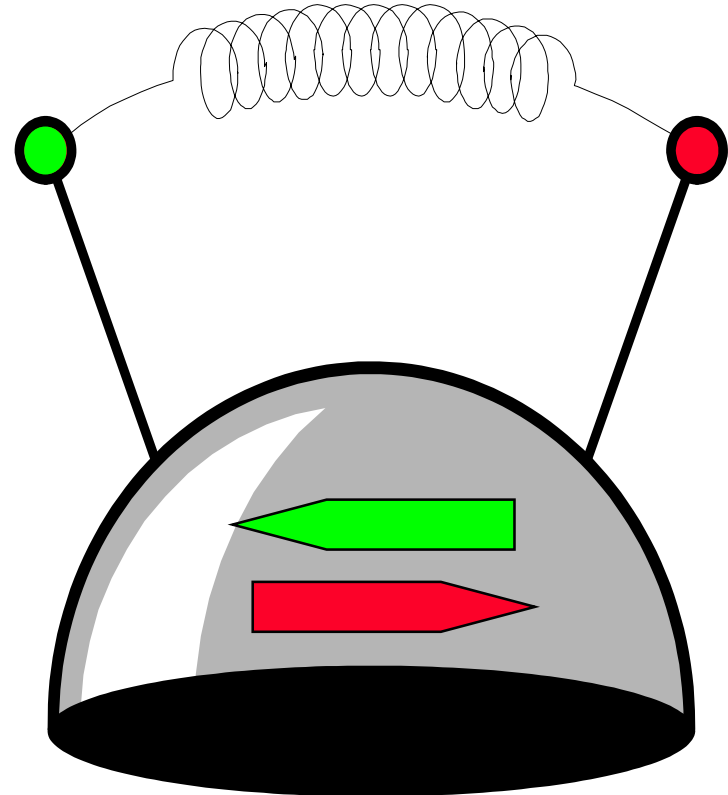
# Tipos Abstratos de Dados

---

# Que Objeto é este?

---

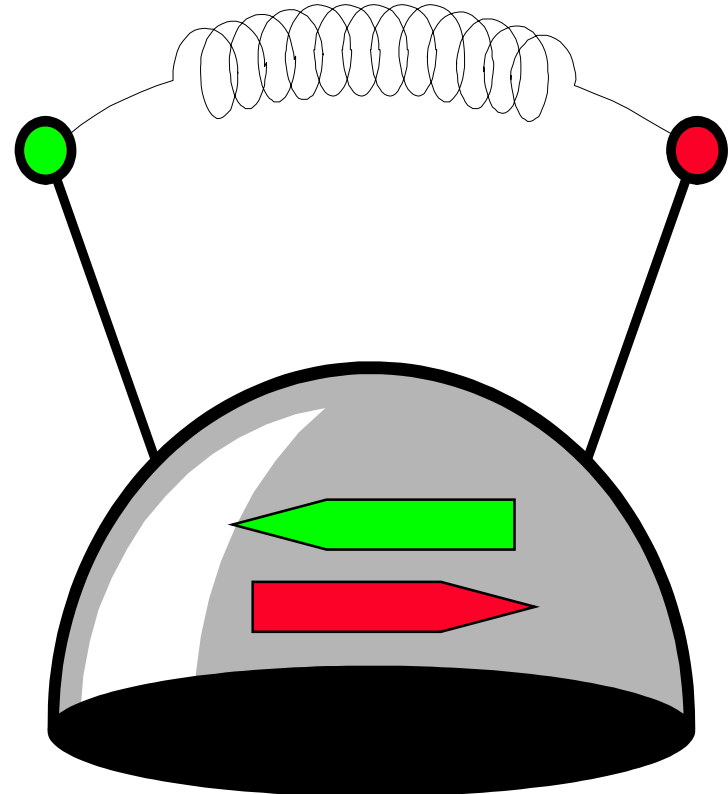
- ❑ Não há uma resposta real para esta questão, mas nós iremos chamá-lo “capacete pensante”
- ❑ A idéia é descrevê-lo por intermédio das **ações** que podem ser realizadas por ele



# Usando os *Slots* do Objeto

---

- ❑ Você pode colocar um pedaço de papel em cada um dos *slots* (setas), com uma sentença escrita sobre cada um
- ❑ Você pode pressionar o botão da esquerda e o capacete pensante dirá a sentença que está sobre a seta verde
- ❑ Idem para o botão direito

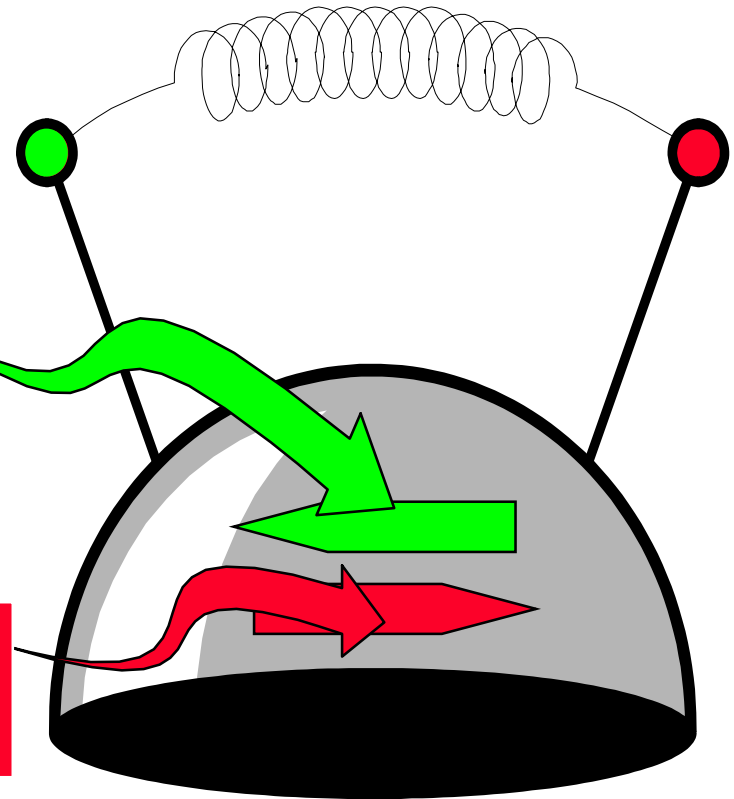


# Exemplo

---

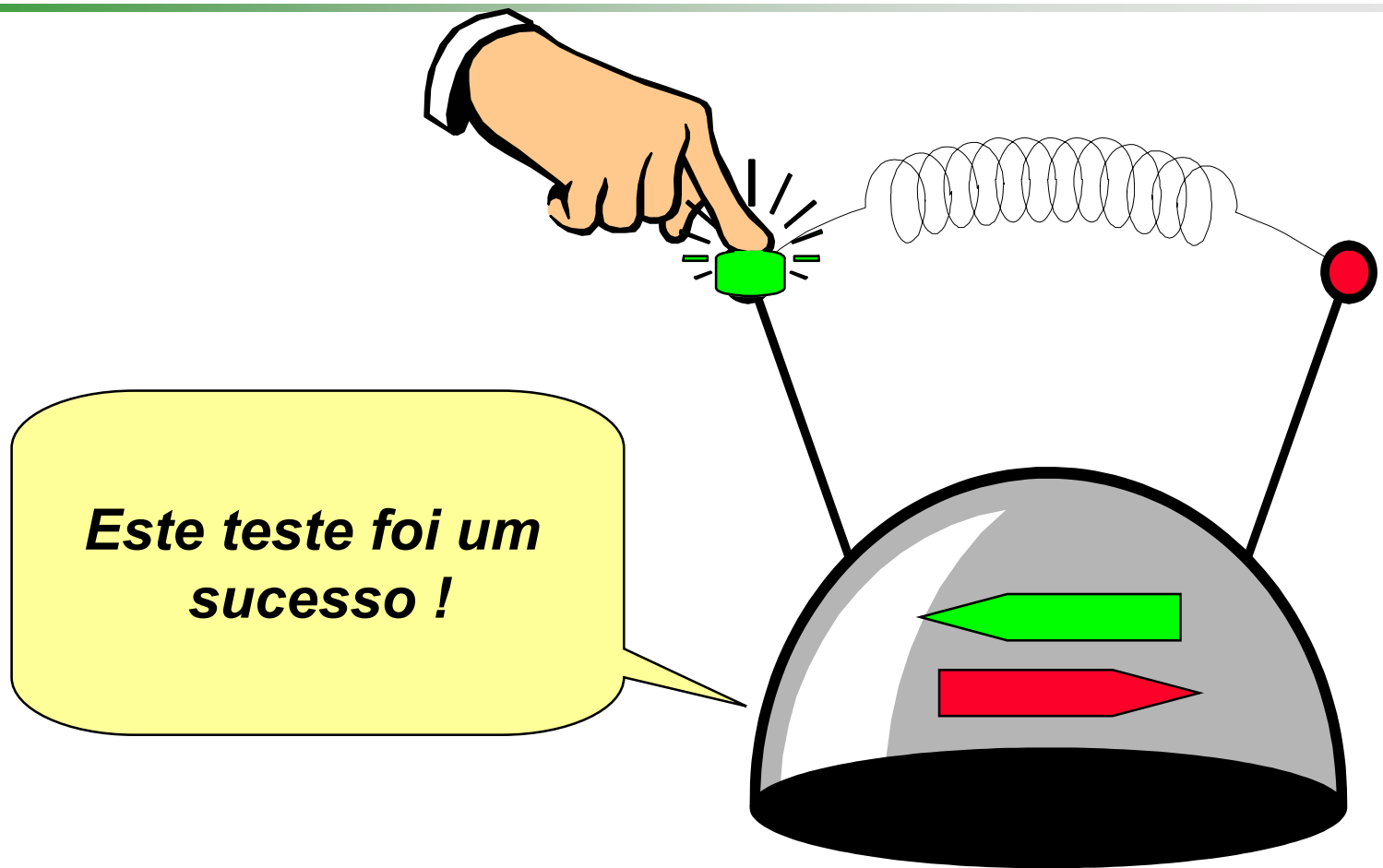
*Este teste foi um sucesso*

*Eu deveria estudar com afinco*



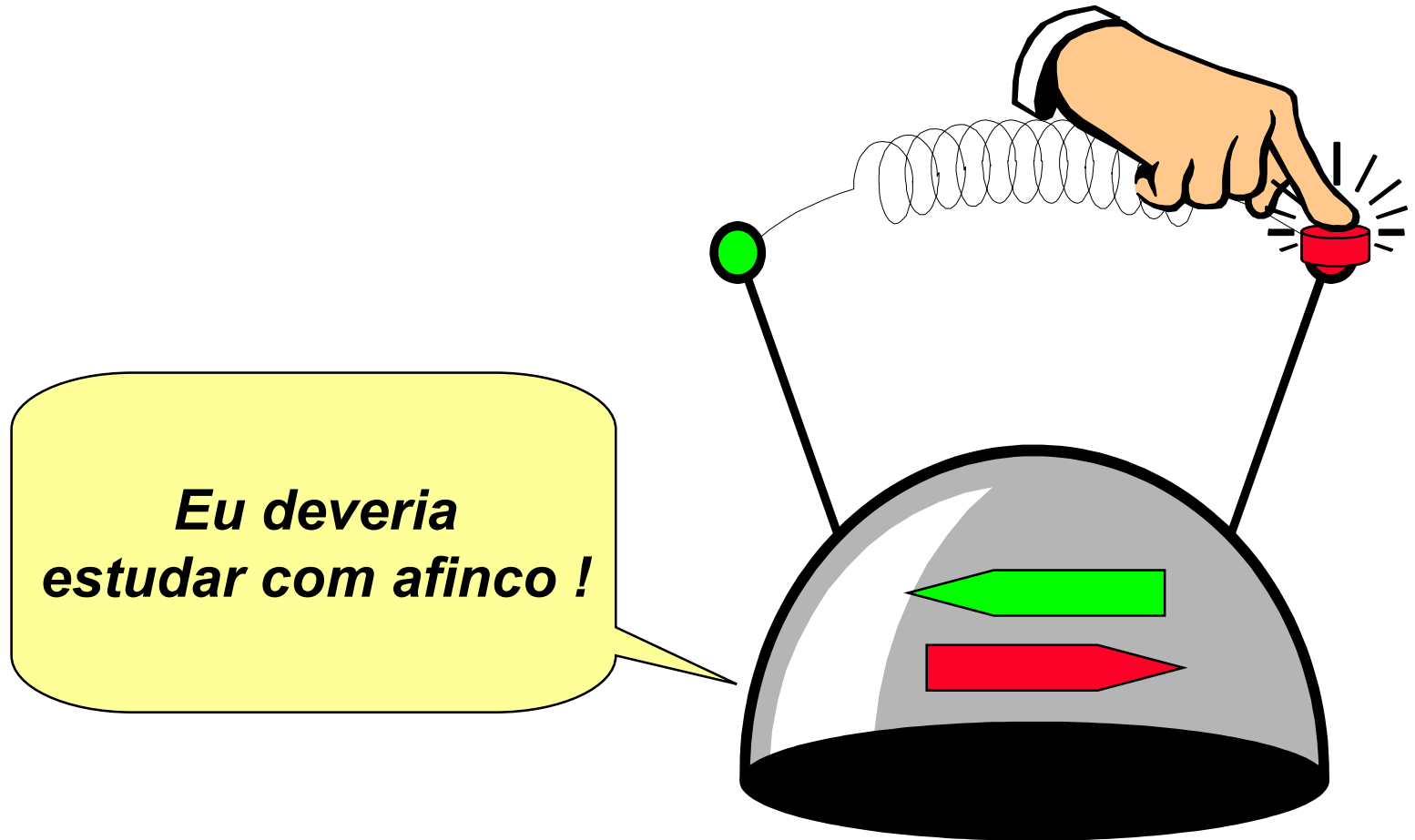
# Exemplo

---

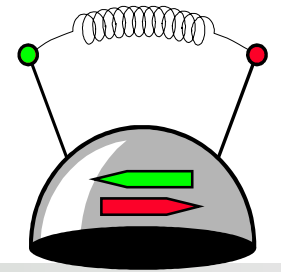


# Exemplo

---



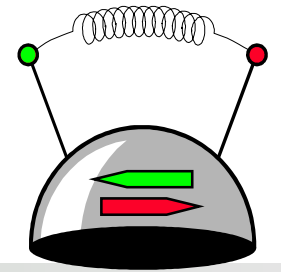
# Implementação do Capacete Pensante



□ Nós podemos implementar o capacete pensante usando uma estrutura similar a um registro (*record* em Pascal ou *struct* em C++)

```
struct ThinkingCap  
{  
    . . .  
};
```

# Implementação do Capacete Pensante

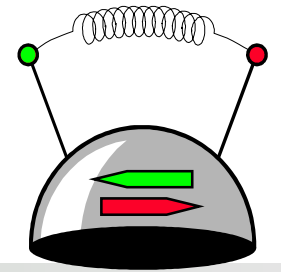


- ❑ Esta estrutura terá dois campos chamados **LeftString** e **RightString**
- ❑ Estes campos são *strings* que irão guardar a informação que for colocada em cada um dos dois *slots*

```
struct ThinkingCap  
{  
    . . .  
    string LeftString;  
    string RightString;  
};
```



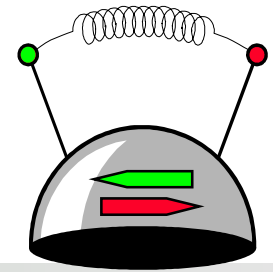
# Implementação do Capacete Pensante



- ❑ Entretanto, em C++ nós usaremos a palavra **class** ao invés da palavra **struct**
- ❑ O uso de um objeto (classe) oferece duas novas características
- ...

```
class ThinkingCap  
{  
    ...  
    string LeftString;  
    string RightString;  
};
```

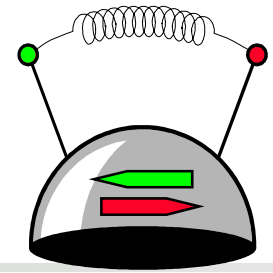
# Implementação do Capacete Pensante



- 1 Os dois campos de dados podem ser declarados como campos privados. Isso restringe o acesso externo aos dados. Somente através de funções e procedimentos do próprio objeto é que se pode manuseá-los

```
class ThinkingCap
{
    . . .
    private:
        string LeftString;
        string RightString;
};
```

# Implementação do Capacete Pensante

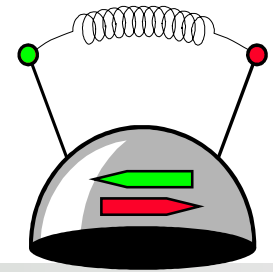


- ② Em um objeto, os procedimentos que o manipulam também são declarados como campos

Os cabeçalhos ou *headers* dos **procedimentos** do capacete pensante são inseridos aqui

```
class ThinkingCap
{
    ...
    private:
        string LeftString;
        string RightString;
};
```

# Implementação do Capacete Pensante

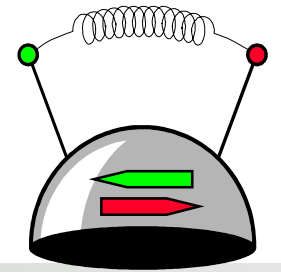


- ② Em um objeto, os métodos também são declarados como campos

Os cabeçalhos ou *headers* dos **métodos** do capacete pensante são inseridos aqui

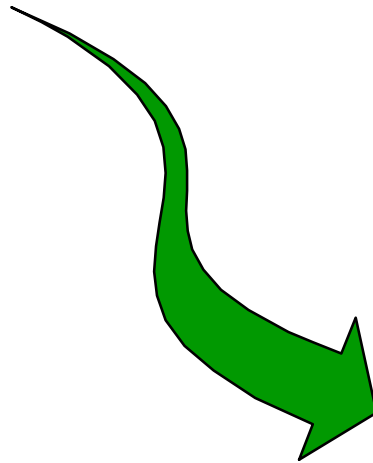
```
class ThinkingCap
{
    ...
    private:
        string LeftString;
        string RightString;
};
```

# Implementação do Capacete Pensante

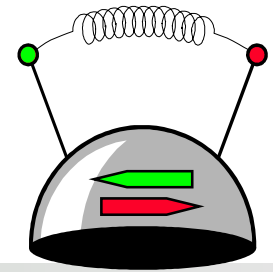


- ❑ O corpo dos métodos do capacete pensante podem aparecer em qualquer lugar depois da declaração de tipos

```
class ThinkingCap
{
    ...
private:
    string LeftString;
    string RightString;
};
```



# Implementação do Capacete Pensante



□ Nosso capacete pensante tem três métodos:

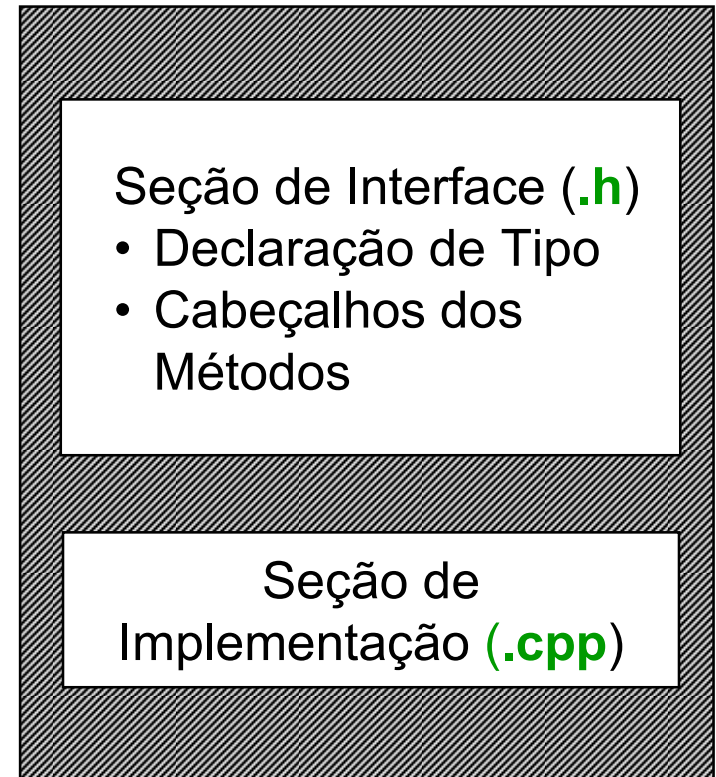
```
class ThinkingCap
{ public:
    void Slots(string NewLeft, string NewRight);
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};
```

O corpo dos métodos  
será colocado abaixo

# Implementação do Capacete Pensante

---

- ❑ Vamos supor que a declaração completa do Capacete Pensante está colocada em um arquivo chamado **Thinker.h**, que nosso programa poderá usar
- ❑ O corpo dos três métodos do Capacete Pensante aparecerá na seção de implementação **Thinker.cpp** que nós escreveremos posteriormente



# Usando o Capacete Pensante

---

- ❑ O programa `// programa Exemplo`  
Exemplo (driver) `#include "Thinker.h"`  
que utiliza o arquivo  
`Thinker.h`



# Usando o Capacete Pensante

---

- ❑ O programa Exemplo irá declarar duas **variáveis** do tipo **ThinkingCap** chamadas **Estudante** e **Festa**

```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

# Usando o Capacete Pensante

---

- ❑ O programa Exemplo irá declarar duas **instâncias** do tipo **ThinkingCap** chamadas **Estudante** e **Festa**

```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

# Usando o Capacete Pensante

---

- ❑ O programa inicia pela chamada ao procedimento Slots de Estudante

```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
    Estudante.Slots("Ola", "Tchau");
```

# Usando o Capacete Pensante

---

- ❑ O programa inicia ativando o método Slots do objeto Estudante

```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
Estudante.Slots("Ola", "Tchau");
```

# Usando o Capacete Pensante

---

- 1 O método de ativação consiste de 4 partes, iniciando com o nome da instância

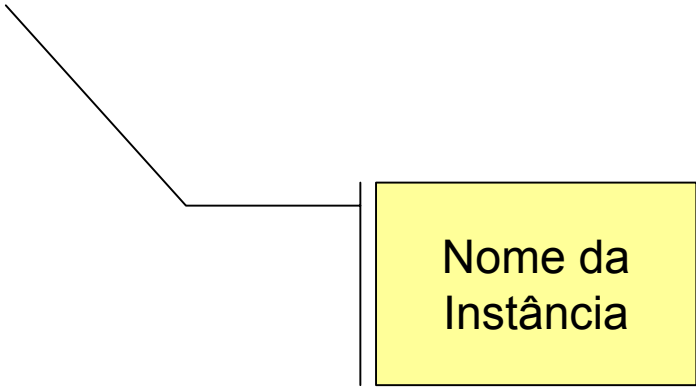
```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
Estudante.Slots("Ola", "Tchau");
```



Nome da  
Instância

# Usando o Capacete Pensante

---

- ② O nome da instância é seguido por um ponto, similar ao modo como qualquer *struct* pode ser seguida por um ponto

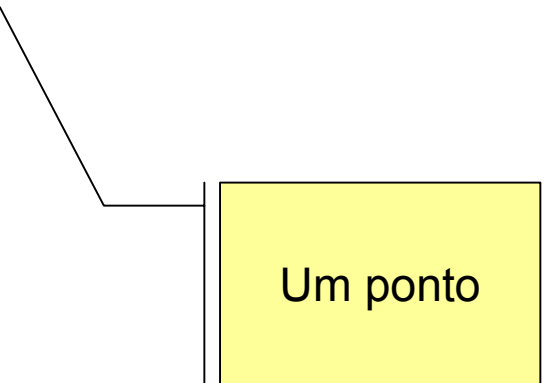
```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
Estudante.Slots("Ola", "Tchau");
```



Um ponto

# Usando o Capacete Pensante

---

③ Depois do ponto vem o nome do método que você está ativando

```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
Estudante.Slots("Ola", "Tchau");
```



Nome do  
Método

# Usando o Capacete Pensante

---

- ④ Finalmente, os **argumentos** para o método. Neste exemplo o primeiro argumento (NewLeft) é "Ola" e o segundo argumento (NewRight) é "Tchau"

```
// programa Exemplo  
#include "Thinker.h"
```

```
int main()  
{ ThinkingCap Estudante, Festa;
```

```
    Estudante.Slots("Ola", "Tchau");
```



Argumentos



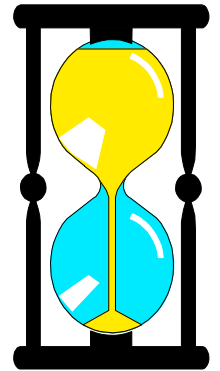
# Um Teste Rápido

---

❑ Como você poderia ativar o método **PressLeft** do objeto Estudante?

```
// programa Exemplo  
#include "Thinker.h"
```

```
int main()  
{ ThinkingCap Estudante, Festa;  
  
  Estudante.Slots("Ola", "Tchau");
```



❑ Qual seria a saída do método **PressLeft** neste ponto do programa?

# Resposta do Teste Rápido

---

- ❑ Note que o método

**PressLeft** não tem argumentos

- ❑ Neste ponto, ativando Estudante.PressLeft() será impressa a string **Ola**

```
// programa Exemplo
```

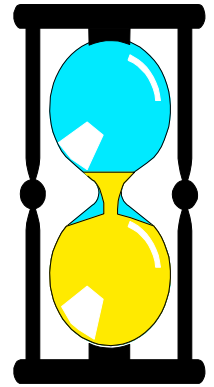
```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
    Estudante.Slots("Ola", "Tchau");
```

```
    Estudante.PressLeft();
```



# Outro Teste Rápido

---

- ❑ Faça o *trace* (rastrear) do programa e diga a saída completa

```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
    Estudante.Slots("Ola", "Tchau");
```

```
    Festa.Slots("Huura!", "Buu!");
```

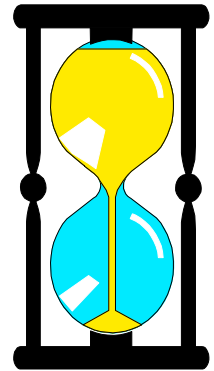
```
    Estudante.PressLeft();
```

```
    Festa.PressLeft();
```

```
    Estudante.PressRight();
```

```
    return 0;
```

```
}
```



# Resposta do Outro Teste Rápido

---

□ **Ola**  
**Huura!**  
**Tchau**

```
// programa Exemplo
```

```
#include "Thinker.h"
```

```
int main()
```

```
{ ThinkingCap Estudante, Festa;
```

```
    Estudante.Slots("Ola", "Tchau");
```

```
    Festa.Slots("Huura!", "Buu!");
```

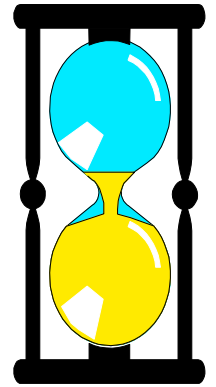
```
    Estudante.PressLeft();
```

```
    Festa.PressLeft();
```

```
    Estudante.PressRight();
```

```
    return 0;
```

```
}
```



# O que você sabe sobre Objetos

---

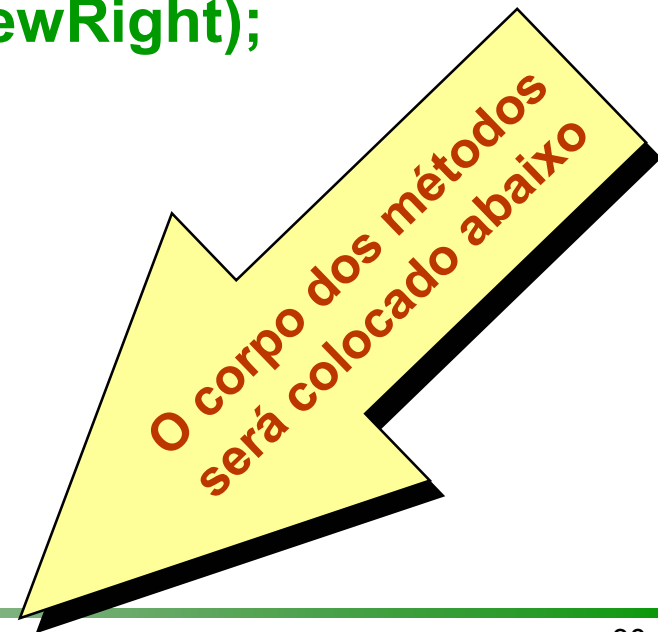
- ✓ Objeto = Dados (Atributos) + Métodos (Funcionalidade) + Encapsulamento
- ✓ Você sabe como declarar um novo tipo de objeto e colocar a declaração em um arquivo `.h`
- ✓ Você sabe como usar o arquivo `.h` em um programa que declara instâncias desse mesmo tipo
- ✓ Você sabe como ativar os métodos
- ✗ Mas você ainda precisa aprender como escrever o corpo dos métodos dos objetos

# Implementação do Capacete Pensante

---

- ❑ Lembre-se que o corpo dos métodos aparece abaixo da declaração de tipo

```
class ThinkingCap
{ public:
    void Slots(string NewLeft, string NewRight);
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};
```

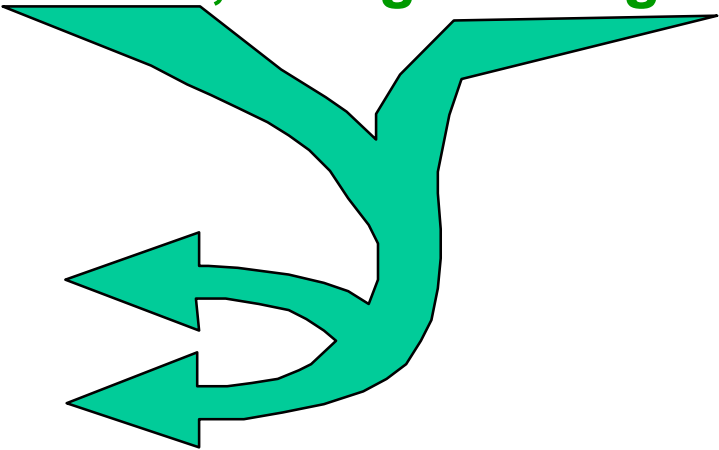


# Implementação do Capacete Pensante

---

- ❑ Nós veremos o corpo de **Slots**, que precisa copiar seus argumentos para os campos de dados privados

```
class ThinkingCap
{ public:
    void Slots(string NewLeft, string NewRight);
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};
```



# Implementação do Capacete Pensante

---

Na maioria das vezes, o corpo dos procedimentos de um objeto não diferem do corpo de qualquer outro procedimento

```
void ThinkingCap::Slots(string NewLeft, string NewRight)
{
    LeftString = NewLeft;
    RightString = NewRight;
}
```

Mas há duas características especiais sobre o corpo dos métodos. . .



# Implementação do Capacete Pensante

---

- ❶ No cabeçalho, o nome dos procedimentos é precedido pelo **nome do objeto**, seguido por **::**, caso contrário o compilador C++ não os consideraria como métodos do objeto

```
void ThinkingCap::Slots(string NewLeft, string NewRight)
{
    LeftString = NewLeft;
    RightString = NewRight;
}
```

# Implementação do Capacete Pensante

---

- ② Dentro do corpo do método, os campos de dados do objeto e outros métodos podem ser utilizados por todos

```
void ThinkingCap::Slots(string NewLeft, string NewRight)
{
    LeftString = NewLeft;
    RightString = NewRight;
}
```

# Implementação do Capacete Pensante

---

- ② Dentro do corpo do método, os campos de dados do objeto e outros métodos podem ser utilizados por todos

```
void ThinkingCap::Slots(string N  
{  
    LeftString = NewLeft;  
    RightString = NewRight;  
}
```

Mas que campos de dados são estes?  
São eles:

- Estudante.LeftString
- Estudante.RightString
- Festa.LeftString
- Festa.RightString ?

# Implementação do Capacete Pensante

---

- ② Dentro do corpo do método, os campos de dados do objeto e outros métodos podem ser utilizados por todos

```
void ThinkingCap::Slots(string N  
{  
    LeftString = NewLeft;  
    RightString = NewRight;  
}
```

Se nós ativamos

**Estudante.Slots:**

Estudante.LeftString

Estudante.RightString

# Implementação do Capacete Pensante

---

- ② Dentro do corpo do método, os campos de dados do objeto e outros métodos podem ser utilizados por todos

```
void ThinkingCap::Slots(string N  
{  
    LeftString = NewLeft;  
    RightString = NewRight;  
}
```

Se nós ativamos

**Festa.Slots:**

Festa.LeftString

Festa.RightString

# Implementação do Capacete Pensante

---

- ❑ Aqui está a implementação do método **PressLeft**, que imprime a mensagem da esquerda

```
void ThinkingCap::PressLeft( )  
{  
    cout << LeftString << endl;  
}
```

# Implementação do Capacete Pensante

---

- ❑ Aqui está a implementação do método **PressLeft**, que imprime a mensagem da esquerda

```
void ThinkingCap::PressLeft( )  
{  
    cout << LeftString << endl;  
}
```

Note como esta implementação do método usa o campo de dados **LeftString** do objeto

# Implementação do Capacete Pensante

---

- ❑ Aqui está a implementação do método **PressRight**, que imprime a mensagem da direita

```
void ThinkingCap::PressRight( )  
{  
    cout << RightString << endl;  
}
```

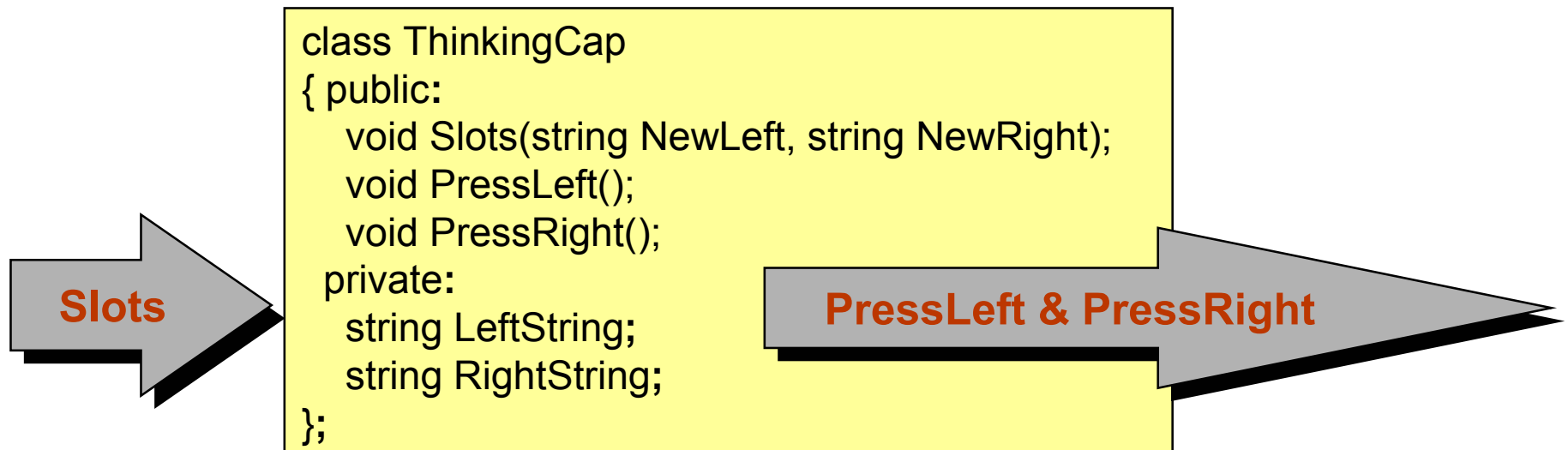
Note como esta implementação do método usa o campo de dados **RightString** do objeto



# Um Padrão Comum

---

- ❑ Frequentemente, um ou mais métodos colocarão dados no campo de dados...



...assim outros métodos podem utilizar os dados

# Um Padrão Comum

---

- Assim, a declaração do objeto é geralmente colocada num arquivo de header (.h) separado...

```
// Thinker.h
// Declaração/interface para capacete pensante
#include <string>
using namespace std;

class ThinkingCap
{ public:
    void Slots(string NewLeft, string NewRight);
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};
```

# Um Padrão Comum

---

## □ ...e a implementação (.cpp) também

```
// Thinker.cpp
// Implementação para capacete pensante
#include <iostream>
#include <string>
#include "Thinker.h"
using namespace std;
void ThinkingCap::Slots(string NewLeft, string NewRight)
{
    LeftString = NewLeft;
    RightString = NewRight;
}
void ThinkingCap::PressLeft( )
{
    cout << LeftString << endl;
}
void ThinkingCap::PressRight( )
{
    cout << RightString << endl;
}
```

# Um Padrão Comum

---

- ❑ Finalmente, os programas que usam o objeto devem incluir o arquivo header do objeto

```
// Programa Exemplo
// Testa objeto capacete pensante
#include "Thinker.h"

int main()
{ ThinkingCap Estudante, Festa;

  Estudante.Slots("Hello", "Goodbye");
  Festa.Slots("Hooray!", "Boo!");
  Estudante.PressLeft( );
  Festa.PressLeft( );
  Estudante.PressRight( );

  return 0;
}
```

# Evitando Múltiplas Inclusões do Arquivo .h

---

- ❑ Quando construimos grandes programas, outras definições e declarações, além dos objetos, são também colocadas em arquivos header (.h)
- ❑ Para evitar que um mesmo arquivo *header* seja incluído várias vezes, é comum usar a seguinte técnica...

```
// Programa 1  
#include "Thinker.h"
```

```
// Programa 2  
#include "Thinker.h"
```

```
// Programa 3  
#include "Thinker.h"
```

# Evitando Múltiplas Inclusões do Arquivo .h

---

```
// Thinker.h
// Declaração/interface para capacete pensante
#include <string>
using namespace std;

#ifdef THINK_H
#define THINK_H

class ThinkingCap
{ public:
    void Slots(string NewLeft, string NewRight);
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};

#endif
```

# Evitando Múltiplas Inclusões do Arquivo .h

```
// Thinker.h
// Declaração/interface para capacete pensante
#include <string>
using namespace std;

#ifdef THINK_H

class ThinkingCap
{ public:
    void Slots(string NewLeft, string
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};

#endif
```

As diretivas do pré-processador evitam que o código entre **#ifdef** e **#endif** seja incluído (em outros programas que usam o objeto) se o nome **THINK\_H** foi definido anteriormente

# Evitando Múltiplas Inclusões do Arquivo .h

```
// Thinker.h
// Declaração/interface para capacete pensante
#include <string>
using namespace std;

#ifndef THINK_H
#define THINK_H

class ThinkingCap
{ public:
    void Slots(string NewLeft, string
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};

#endif
```

Se o header não foi incluído anteriormente (no programa que usa o objeto), o nome **THINK\_H** é definido pela diretiva **#define** e código entre **#ifndef** e **#endif** é incluído no programa



# Evitando Múltiplas Inclusões do Arquivo .h

```
// Thinker.h
// Declaração/interface para capacete pensante
#include <string>
using namespace std;
```

```
#ifndef THINK_H
#define THINK_H
```

```
class ThinkingCap
{ public:
    void Slots(string NewLeft, string
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};
```

```
#endif
```

Se o header foi incluído anteriormente (no programa que usa o objeto), o nome **THINK\_H** foi definido anteriormente e o código entre **#ifndef** e **#endif** não é incluído novamente no programa

# Evitando Múltiplas Inclusões do Arquivo .h

```
// Thinker.h
// Declaração/interface para capacete pensante
#include <string>
using namespace std;

#ifndef THINK_H
#define THINK_H

class ThinkingCap
{ public:
    void Slots(string NewLeft, string
    void PressLeft();
    void PressRight();
private:
    string LeftString;
    string RightString;
};

#endif
```

Geralmente, o nome a ser definido é o nome do arquivo *header* (em letras maiúsculas), substituindo o ponto por sublinhado:

Exemplo:

Arquivo header: **Think.h**

Nome definido: **THINK\_H**

# Pontos Importantes

---

- ❑ Objetos são como registros (*structs*) com “campos” extras que são os métodos
- ❑ Você deve saber como declarar um novo tipo de objeto, como implementar seus tipos e como usá-lo
- ❑ Frequentemente, os métodos de um objeto
  - colocam informações nos campos de dados ou
  - usam as informações contidas nos mesmos

# Tipos Abstratos de Dados

---

- ❑ Os **Tipos Abstratos de Dados** também denominados *Abstract Data Types* (ADT), consistem em uma forma de definir um novo tipo de dado juntamente com as operações que manipulam este novo tipo
- ❑ O Capacete Pensante, por exemplo, é um ADT
- ❑ Veremos um outro exemplo, o ADT Sacola

# Sacolas (Bags)

---

- No nosso primeiro exemplo, pense em uma sacola



# Sacolas (Bags)

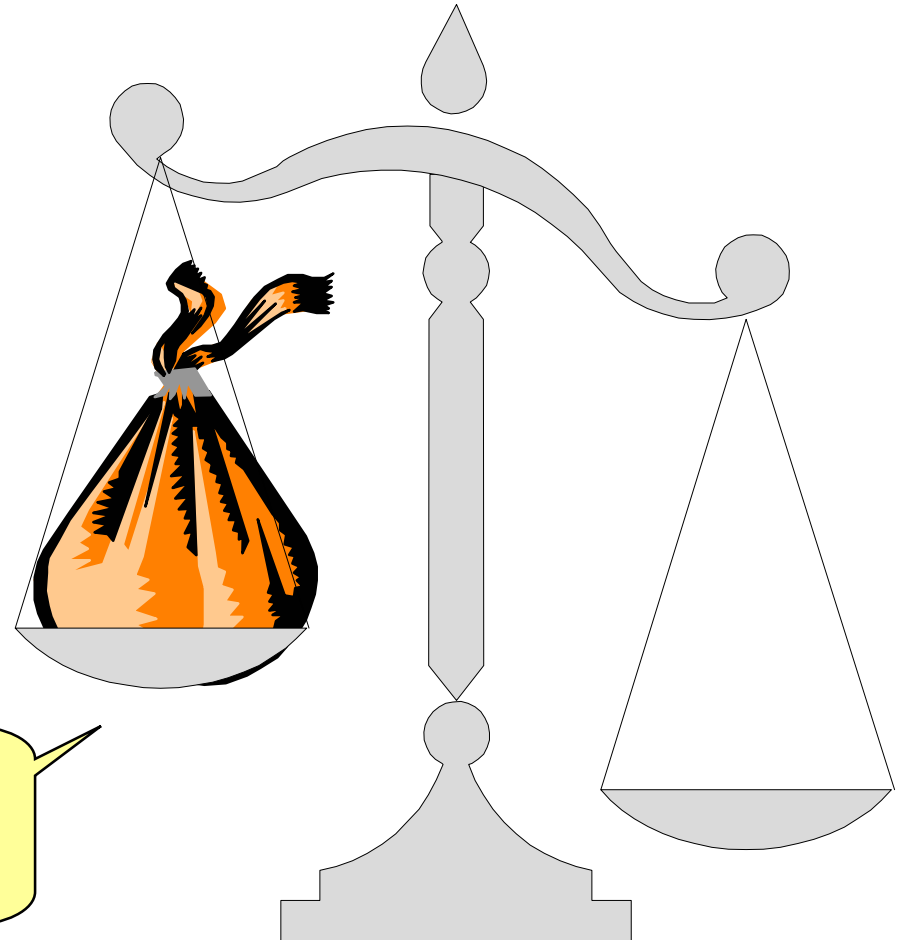
---

- ❑ No nosso primeiro exemplo, pense em uma sacola
- ❑ Dentro desta sacola estão alguns números



# Estado Inicial de uma Sacola

- ❑ Quando você começar a usar uma sacola, ela estará vazia
- ❑ Nós assumimos isto como sendo o **estado inicial** de qualquer sacola que seja usada



*Esta sacola  
está vazia!*

# Inserindo Números em uma Sacola

---

- ❑ Números podem ser **inseridos** em uma sacola





# Inserindo Números em uma Sacola

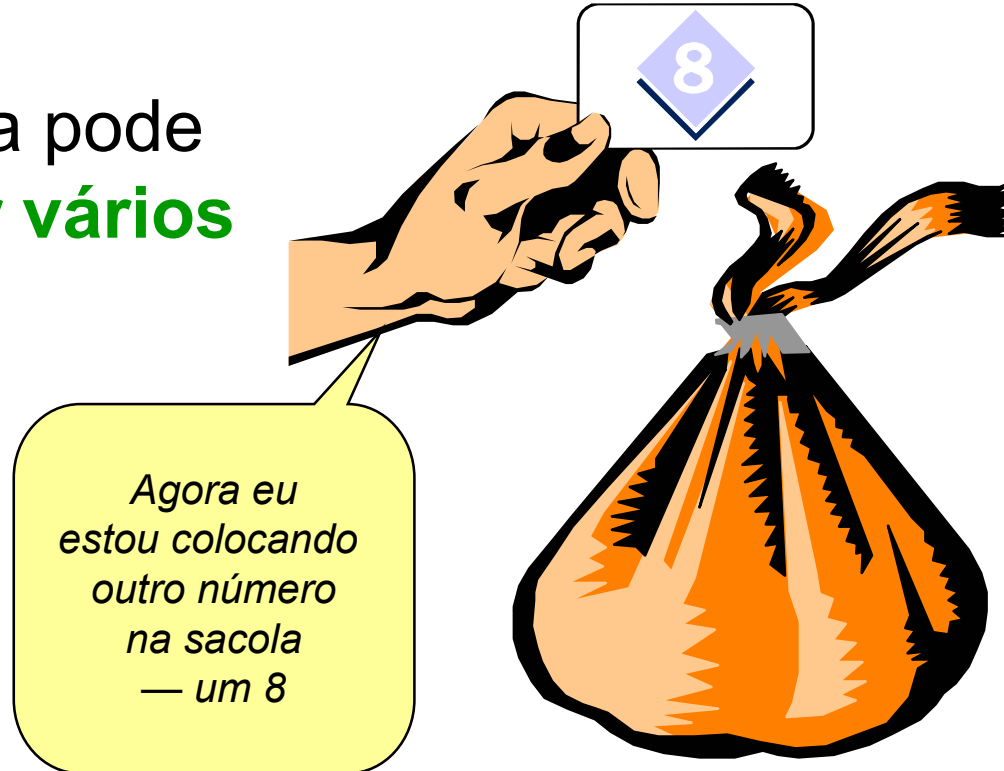
---

- Números podem ser **inseridos** em uma sacola



# Inserindo Números em uma Sacola

- ❑ Números podem ser **inseridos** em uma sacola
- ❑ Uma sacola pode **armazenar vários números**



# Inserindo Números em uma Sacola

- ❑ Números podem ser **inseridos** em uma sacola
- ❑ Uma sacola pode **armazenar vários números**



# Inserindo Números em uma Sacola

- ❑ Números podem ser **inseridos** em uma sacola
- ❑ Uma sacola pode **armazenar vários números**
- ❑ Nós podemos até mesmo inserir o **mesmo número mais de uma vez**



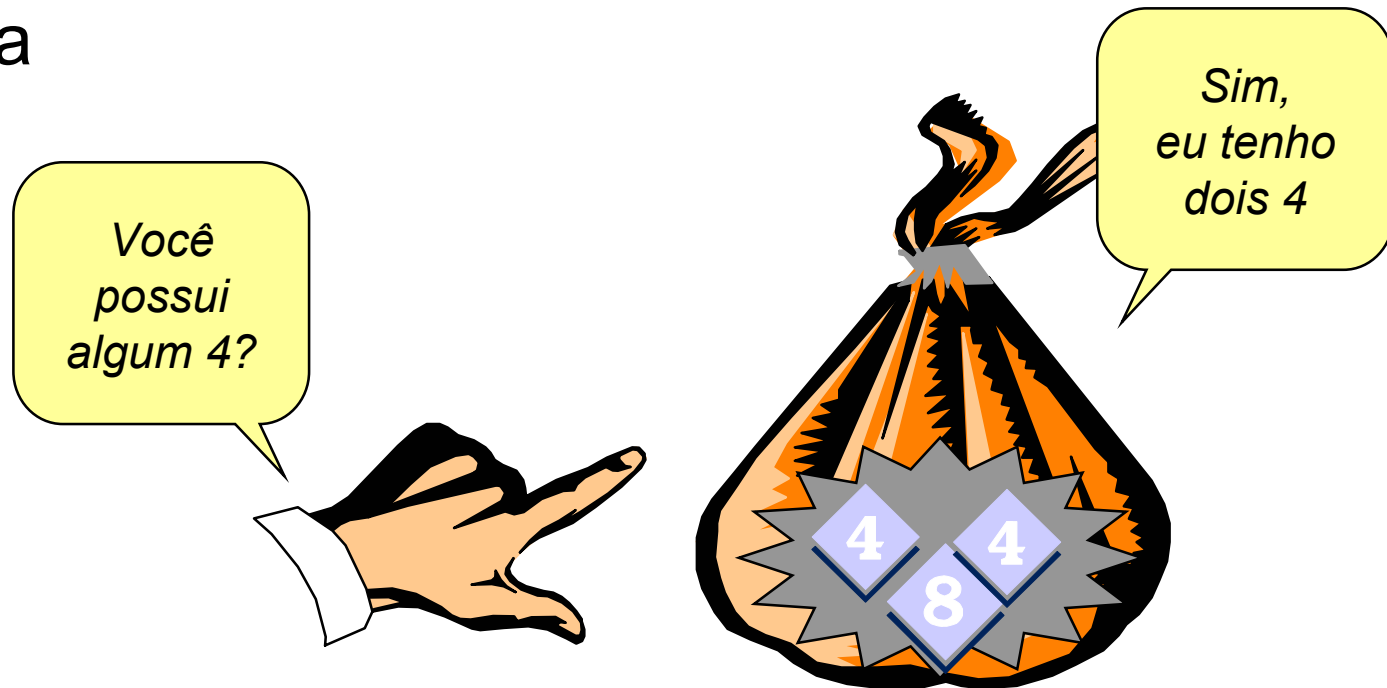
# Inserindo Números em uma Sacola

- ❑ Números podem ser **inseridos** em uma sacola
- ❑ Uma sacola pode **armazenar vários números**
- ❑ Nós podemos até mesmo inserir o **mesmo número mais de uma vez**



# Examinado uma Sacola

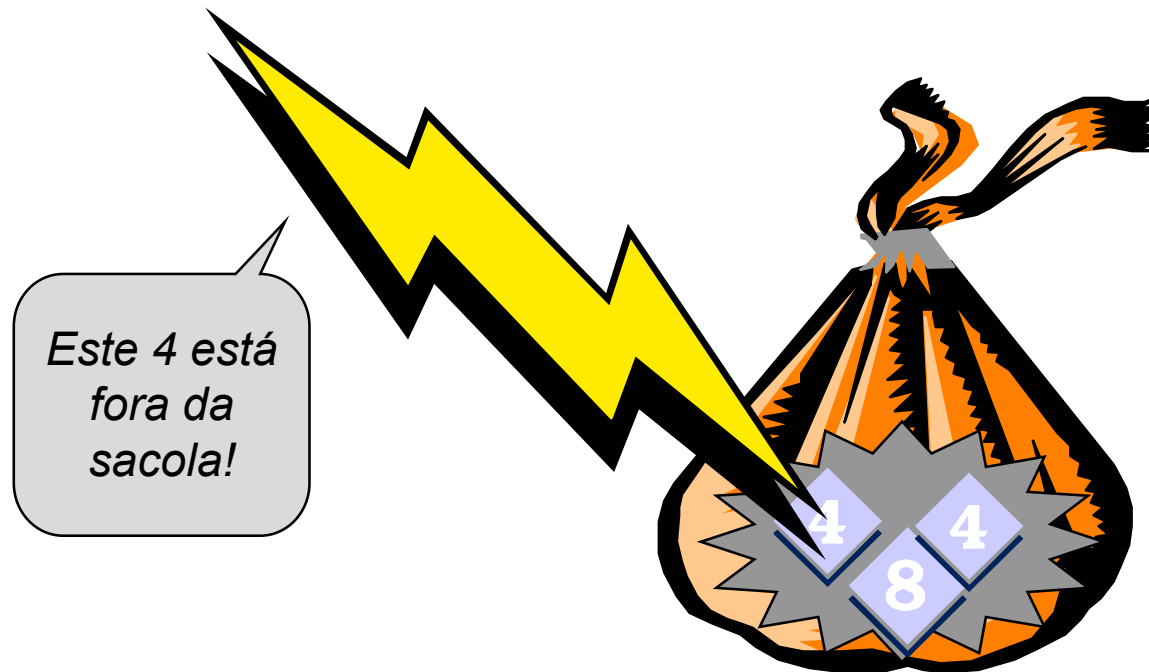
- Nós podemos **perguntar a respeito do conteúdo** de uma sacola



# Apagando um Número de uma Sacola

---

- Nós podemos **apagar** um número de uma sacola



# Apagando um Número de uma Sacola

- ❑ Nós podemos **apagar** um número de uma sacola
- ❑ Porém nós apagamos apenas um número por vez





# Verificando se a Sacola está Cheia

---

- Uma outra operação é verificar se uma sacola possui **espaço para outros números**



# Resumo das Operações sobre Sacolas

---

1. Uma sacola pode ser colocada em seu estado inicial, que corresponde a uma sacola vazia
2. Números podem ser inseridos na sacola
3. Você pode verificar quantas ocorrências de um certo número estão na sacola
4. Números podem ser apagados da sacola
5. Você pode verificar se uma sacola está cheia

# O ADT Sacola

---

- Uma Sacola pode ser implementada em uma linguagem de programação como C++

# O ADT Sacola

---

- ❑ Uma Sacola pode ser implementada em uma linguagem de programação como C++
- ❑ Como um ADT, a implementação inclui:
  - Uma declaração de tipo

```
class Bag
{ public:
    . . .
    private:
        . . .
};
```

# O ADT Sacola

---

- ❑ Uma Sacola pode ser implementada em uma linguagem de programação como C++
- ❑ Como um ADT, a implementação inclui:
  - Uma declaração de tipo
  - Cinco funções ou procedimentos para implementar as cinco operações

```
class Bag
{   public:
    Bag( ...
    void Insert( ...
    int Occurrence(
    void Remove( ...
    bool Full ( ...
    private:
        . . .
};
```

# O Procedimento de Inicialização

---

- ❑ Colocar uma sacola em seu estado inicial (uma sacola vazia)

```
Bag::Bag() // construtor para ADT Bag
// Pré-condição: Nenhuma
// Pós-condição: A sacola é iniciada vazia
{
    ...
}
```

# O Procedimento de Inicialização

---

- ❑ Colocar uma sacola em seu estado inicial (uma sacola vazia)

```
Bag::Bag() // construtor para ADT Bag
```

```
// Pré-condição: Nenhuma
```

```
// Pós-condição:
```

```
{
```

```
    ...
```

```
}
```

O construtor de um ADT em C++ tem sempre o mesmo nome do ADT (classe) e é executado automaticamente quando se declara variáveis do ADT definido

# O Procedimento de Inicialização

- ❑ Colocar uma sacola em seu estado inicial (uma sacola vazia)

```
Bag::Bag() // construtor para ADT Bag
```

```
// Pré-condição: Nenhuma
```

```
// Pós-con
```

```
{
```

```
...
```

```
}
```

Note que um construtor não possui tipo associado, nem mesmo *void*



# O Procedimento de Inserção

---

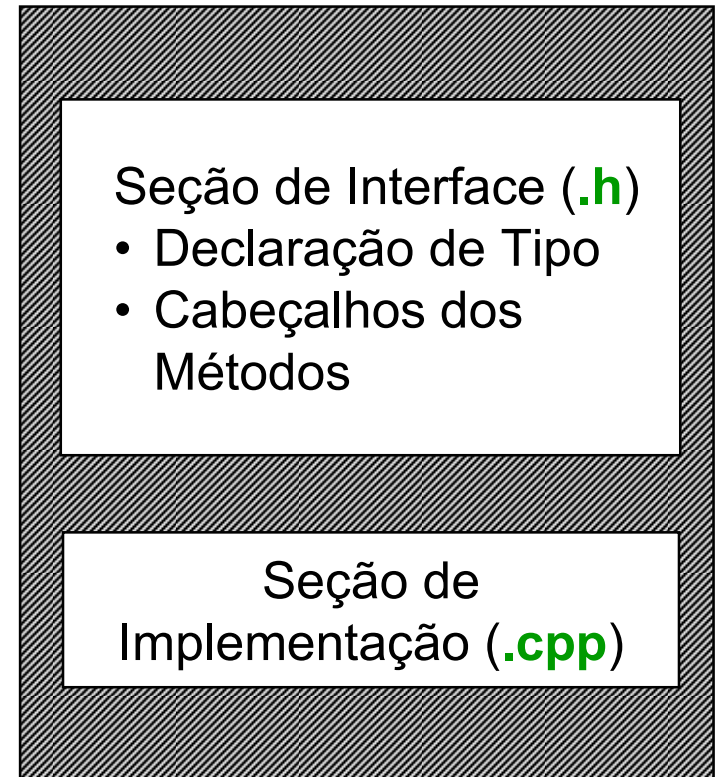
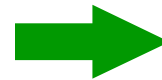
- ❑ Inserir um novo número na sacola

```
void Bag::Insert(int NewEntry)
// Pré-condição: Bag já esteja previamente
// inicializada e não esteja cheia
// Pós-condição: Uma nova cópia de NewEntry
// é adicionada à sacola
{
    ...
}
```

# Arquivos para o ADT Sacola

---

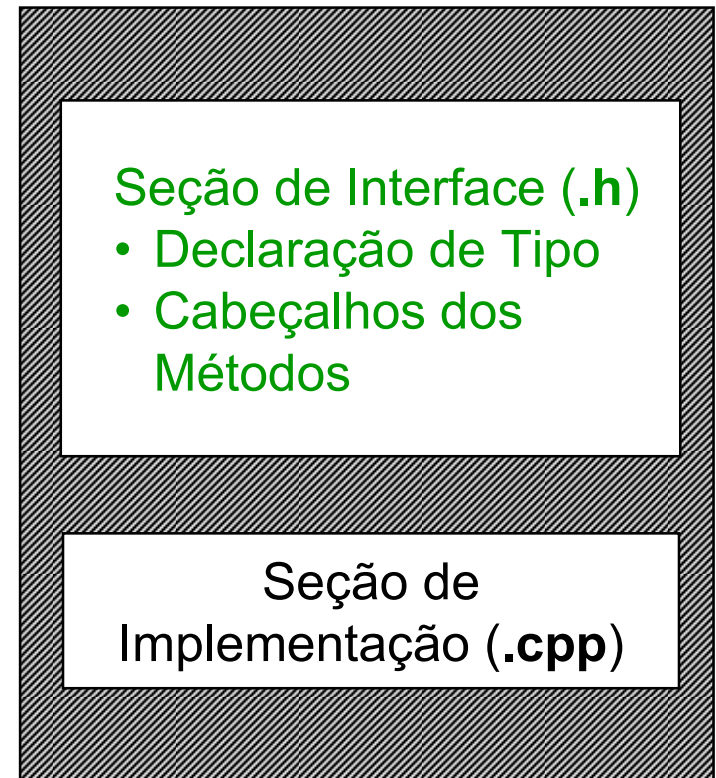
- ❑ É conveniente que o ADT sacola seja colocado em dois arquivos separados, desta forma qualquer programador pode usar o novo tipo sacola
- ❑ Nós vamos olhar as **partes do ADT sacola**



# Arquivos para o ADT Sacola

---

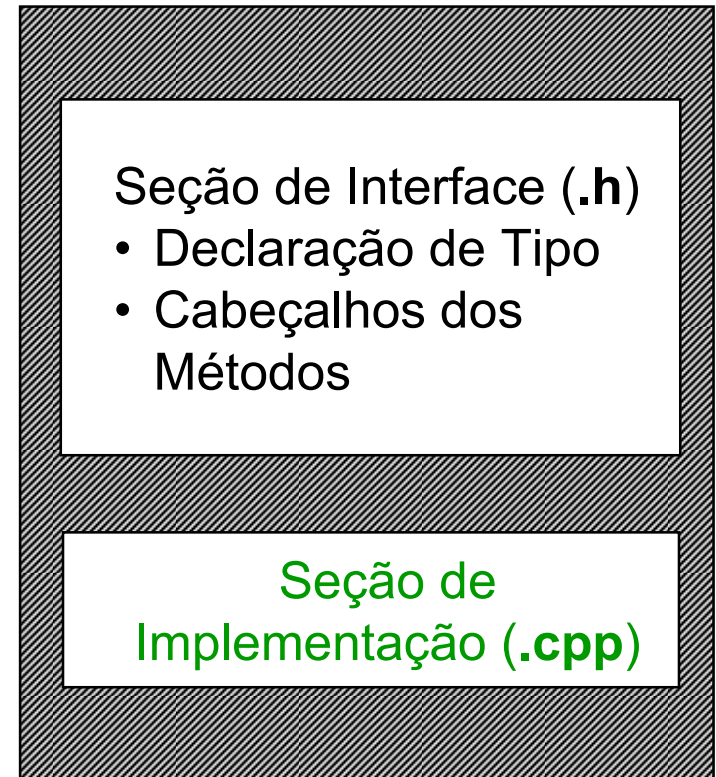
- ❑ Existe uma seção chamada de **especificação** ou **interface** do ADT
- ❑ Esta seção lista o nome do novo tipo de dado (**Sacola**) e também fornece **cabeçalhos e especificações** para as cinco operações sobre a sacola
- ❑ Procedimentos e dados podem ser públicos ou privados



# Arquivos para o ADT Sacola

---

- ❑ Na seção de **implementação** são colocados os procedimentos e funções (métodos) que implementam o ADT Sacola
- ❑ Seguem os **cabeçalhos + corpos** das cinco operações
- ❑ Cada cabeçalho deve ser precedido pelo nome do ADT seguido de **::**



# Questão

---

Suponha que um Benfeitor Misterioso forneça a você um ADT Sacola, mas você pode apenas ler a especificação do ADT. Você não pode ler como o ADT foi implementado. **Você pode escrever um programa que utiliza este ADT sacola?**

- ❶ Sim
- ❷ Não. Não sem poder ler a seção de implementação para o tipo sacola
- ❸ Não. Não sem poder ler a seção de declaração para o tipo sacola e também ver a seção de implementação

# Questão

---

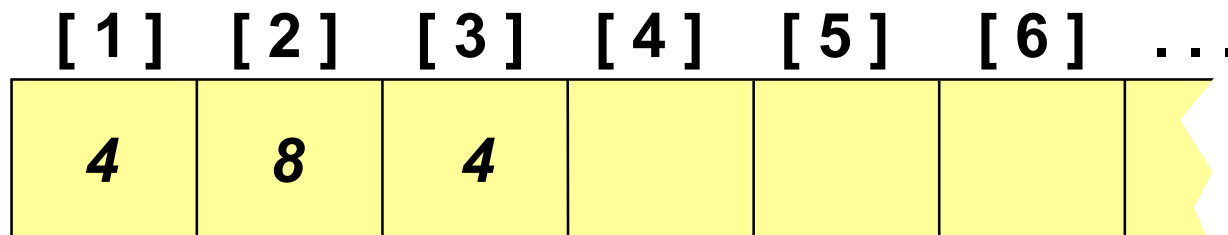
Suponha que um Benfeitor Misterioso forneça a você um ADT Sacola, mas você pode apenas ler a especificação do ADT. Você não pode ler como o ADT foi implementado. **Você pode escrever um programa que utiliza este ADT sacola?**

**1 Sim**

Você sabe o nome do novo tipo de dado, o que é suficiente para você declarar variáveis do tipo sacola. Você também sabe os cabeçalhos e especificações para cada operação

# Detalhes de Implementação

- As entradas em uma sacola serão armazenadas no **início de um vetor**, como mostra este exemplo

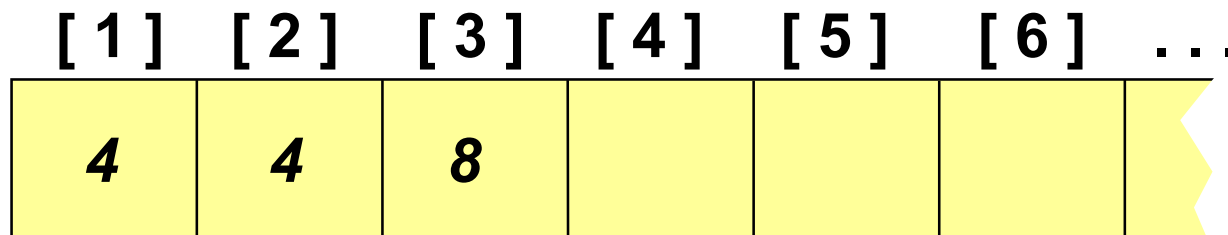


Um vetor de inteiros

Nós não nos interessamos para o que está armazenado nesta parte do vetor

# Detalhes de Implementação

- As entradas podem aparecer em qualquer ordem. A figura abaixo representa a mesma sacola que a anterior...



Um vetor de inteiros

Nós não nos interessamos para o que está armazenado nesta parte do vetor



# Detalhes de Implementação

- ... e esta também representa a mesma sacola



[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]	...
8	4	4				

Um vetor de inteiros

Nós não nos interessamos para o que  
está armazenado nesta parte do vetor

# Detalhes de Implementação

- ❑ Nós precisamos também armazenar quantos números estão na sacola

**3**

Um inteiro armazena o tamanho da sacola



[ 1 ]	[ 2 ]	[ 3 ]	[ 4 ]	[ 5 ]	[ 6 ]	...
8	4	4				

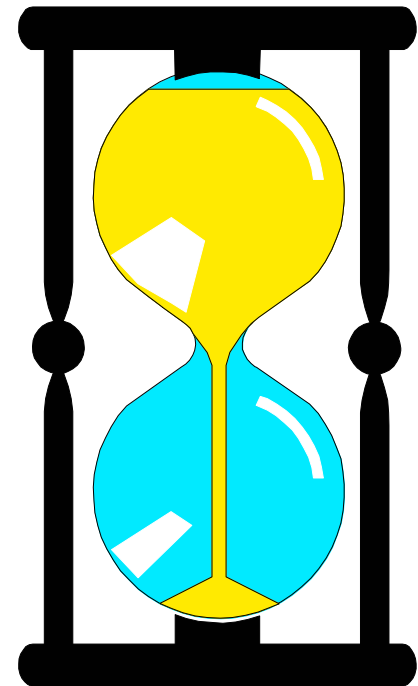
Um vetor de inteiros

Nós não nos interessamos para o que está armazenado nesta parte do vetor

# Questão

---

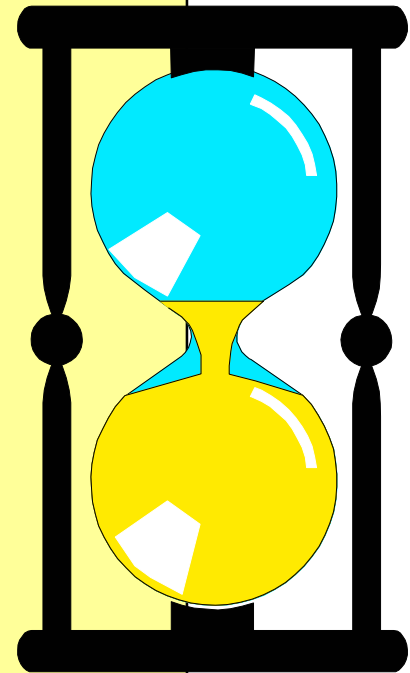
❑ Utilize estas idéias para escrever uma declaração de tipo que poderia implementar o tipo de dado sacola. A declaração deve ser um objeto com dois campos de dados. Faça uma sacola capaz de armazenar 20 inteiros



# Uma Solução

---

```
const int CAPACITY = 20;  
class Bag  
{ public:  
    Bag( ...  
    void Insert( ...  
    int Occurrence(  
    void Remove( ...  
    bool Full ( ...  
private:  
    int Data[CAPACITY+1];  
    int Used;  
};
```

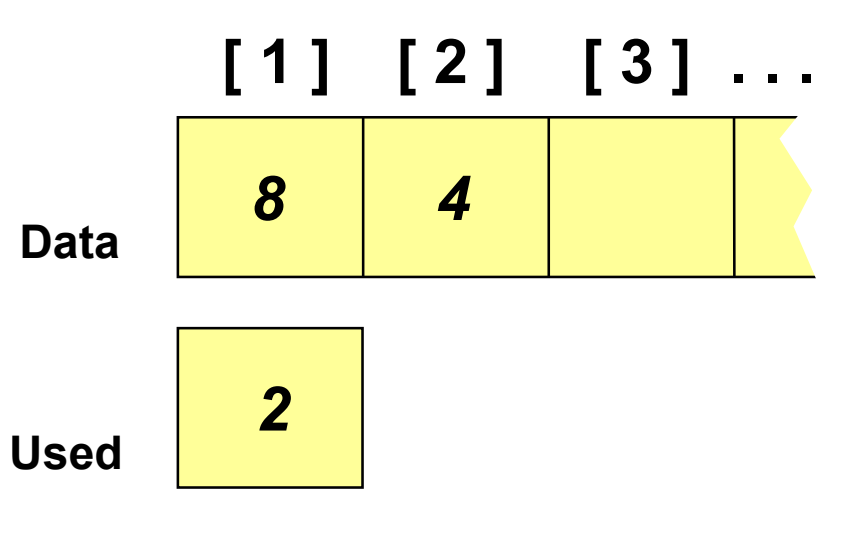


# Um Exemplo de Chamada a Insert

---

```
void Bag::Insert(int NewEntry)
```

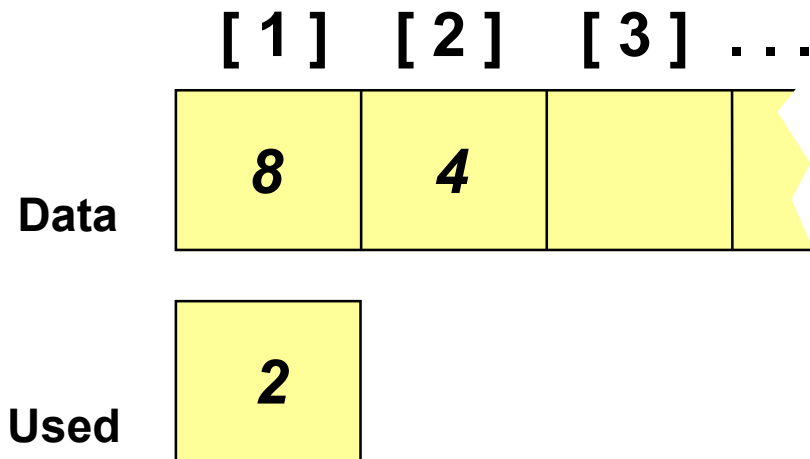
Antes de chamar **Insert**,  
nós supomos ter uma sacola B:



# Um Exemplo de Chamada a Insert

```
void Bag::Insert(int NewEntry)
```

Nós fazemos uma chamada a  
**B.Insert(17)**

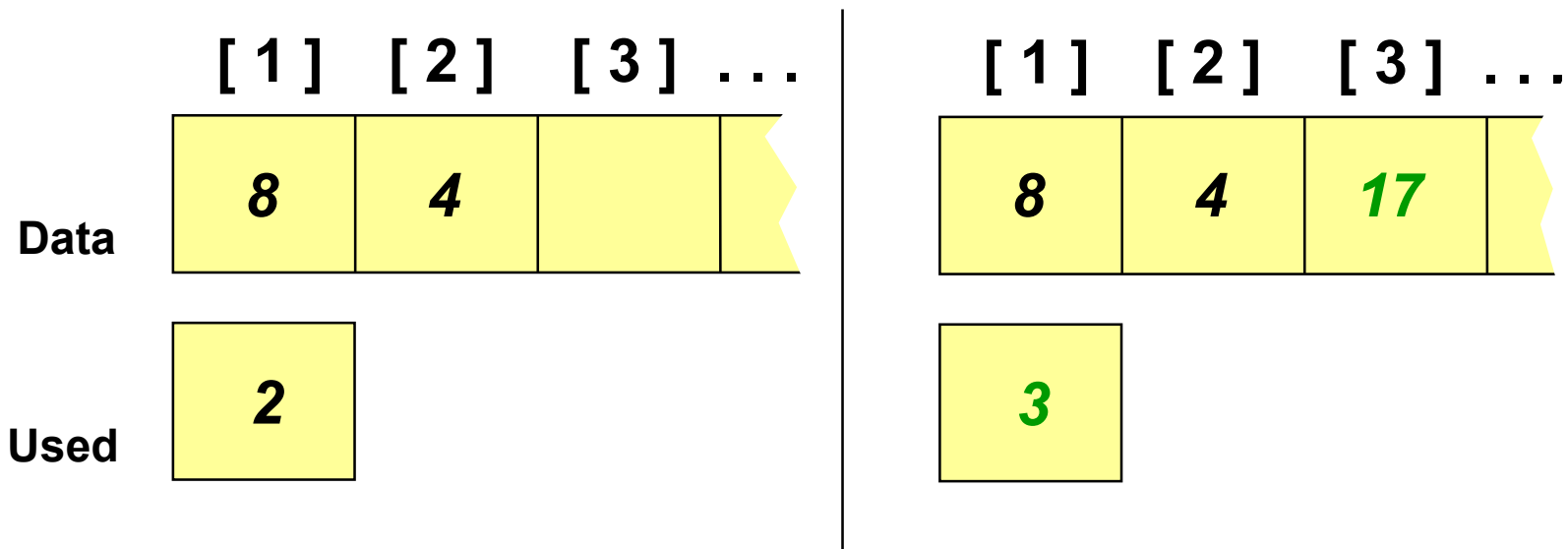


Quais valores serão armazenados em **Data** e **Used** depois que a chamada de procedimento termina?

# Um Exemplo de Chamada a Insert

```
void Bag::Insert(int NewEntry)
```

Depois da chamada a **B.Insert(17)**,  
nós teremos esta sacola B:



# Pseudo-código para Insert

---

- ❶ Se a sacola está cheia (**Used** = **CAPACITY**) então imprima uma mensagem de erro e aborte
- ❷ Adicione um a **Used**
- ❸ Coloque **NewEntry** na posição apropriada de **Data**

Qual é a “posição apropriada” de **Data**?



# Pseudo-código para Insert

---

- ❶ Se a sacola está cheia (**Used** = **CAPACITY**) então imprima uma mensagem de erro e aborte
- ❷ Adicione um a **Used**
- ❸ Coloque **NewEntry** na posição apropriada de **Data**

```
Data[ Used ] = NewEntry;
```

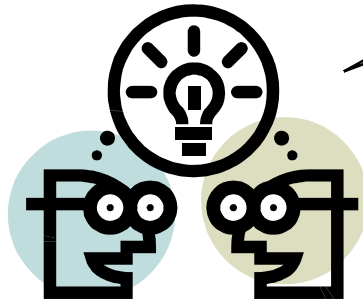
# Outras Operações de Sacola

---

- ❑ Lembre-se: Se você está somente usando o ADT sacola, então você não precisa saber como as operações foram implementadas
- ❑ Mais tarde você poderá **reimplementar** a sacola usando um algoritmo mais eficiente. Mas os **programas que utilizam a sacola não terão que ser alterados**

# Uma Grande Idéia

---



*Nós temos que ser capazes de alterar a implementação de um ADT sem ter que alterar os programas que utilizam este ADT*

- ❑ Mais tarde você poderá **reimplementar** a sacola usando um algoritmo mais eficiente
- ❑ Entretanto, os **programas que utilizam a sacola não terão que ser alterados**

# Outros Tipos de Sacolas

---

- ❑ Neste exemplo, nós implementamos uma sacola contendo **inteiros**
- ❑ Mas nós poderíamos ter uma sacola de **números reais**, uma sacola de **caracteres**, uma sacola de **strings**...
- ❑ Exercício: Suponha que você deseje uma destas implementações. O que você terá a modificar na implementação atual?

# Resumo

---

- ❑ Um ADT é um novo tipo de dado, juntamente com operações que o manipulam, tal como o ADT de sacola e suas cinco operações
- ❑ O ADT é colocado em dois arquivos (**.h** e **.cpp**)
- ❑ Qualquer programa pode usar o ADT
- ❑ Se a implementação for modificada, os programas que utilizam o ADT não terão que ser alterados
- ❑ ADT genéricos são um tópico importante que será abordado no restante do curso