> “Um mouse é um dispositivo que contém um, dois, ou três botões, dependendo da estimativa que os projetistas dão para a capacidade intelectual de seus usuários” (Tanenbaum, Bos; 2016).

# Introdução a STL

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

# Standard Template Library

A STL (Standard Template Library) contém diversas implementações já prontas.

Pertence a **biblioteca padrão** do C++.

Algoritmos **eficientes, genéricos e retrocompatíveis** para:

Listas encadeadas, vetores redimensionáveis, mapas hash, árvores, ...

Evita a "reinvenção da roda" toda vez que um programa precisa ser feito.

Veja em www.cplusplus.com/reference/stl.

# STL

Vamos usar a classe `list` da STL como exemplo.

É comumente implementada como uma **lista duplamente encadeada**.

Veja as funções membro da classe list em www.cplusplus.com/reference/list/list.

# Instanciando

Inclua o header da estrutura desejada.

Para list.

```
#include<list>
```

Para criar uma instância.

```
std::list<TIPO_ITENS_LISTA> minhaLista;
```

Você pode criar uma lista de qualquer tipo. Inteiro, Pessoa, ponteiros para Disciplina, ...

Para tornar isso possível, a STL utiliza polimorfismo paramétrico, que estudaremos no futuro.

# Exemplos de funções membro de list

`push_back`

Adiciona o elemento no final da lista.

`push_front`

Adiciona o elemento no início da lista.

`remove`

Remove determinado elemento da lista.

`empty`

Retorna verdadeiro se a lista está vazia, ou falso caso contrário.

# Exemplo

```cpp
#include<list>

int main(){
   std::list<int> lista;

   lista.push_back(10);
   lista.push_back(20);
   lista.push_front(30);

   return 0;
}
```

# Acessando os elementos

Para acessar os elementos da lista, precisamos de um **iterador** (**iterator**).

Padrão de Projeto.

Por enquanto, veremos apenas como **usar** um iterador.

**Iterador**: objeto capaz de percorrer um container (lista, vetor, árvore, ...).

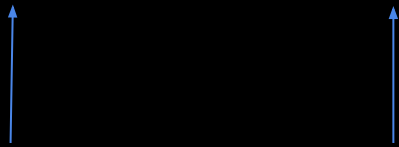
O iterador aponta para o elemento atual.

Pode receber o próximo elemento.

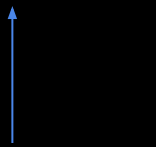
Criamos um loop, onde fazermos o iterador apontar para o próximo elemento a cada iteração, até terminar a lista.

# Declarando um iterador

```
std::TIPO_CONTAINER<TIPO_OBJETOS_CONT>::iterator meuIterador;
```



list, vector, map, ...



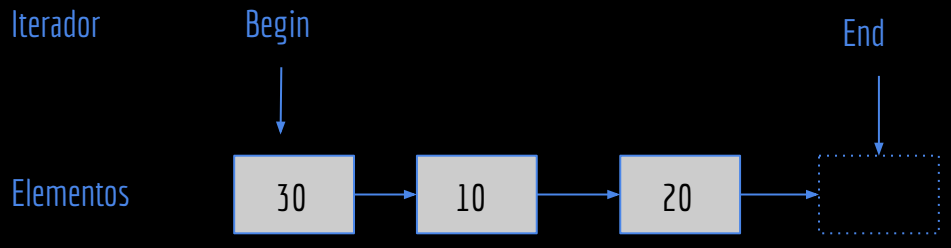
double, int, Pessoa, ...

# Exemplo

```
#include<iostream>
#include<list>

int main(){
   std::list<int> lista;
   lista.push_back(10);
   lista.push_back(20);
   lista.push_front(30);

   std::list<int>::iterator it;
   for(it = lista.begin(); it != lista.end(); ++it)
       std::cout << *it << std::endl;

   return 0;
}
```

Declarado um iterador capaz de iterar em listas de inteiros.

# Exemplo

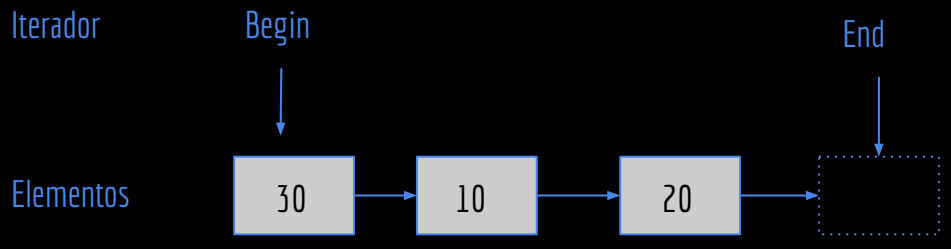



Adaptado de Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.

```cpp
#include<iostream>
#include<list>

int main(){
    std::list<int> lista;
    lista.push_back(10);
    lista.push_back(20);
    lista.push_front(30);

    std::list<int>::iterator it;
    for(it = lista.begin(); it != lista.end(); ++it)
        std::cout << *it << std::endl;

    return 0;
}
```

Todos containers iteráveis possuem uma função begin() que retorna um iterador para o início do container.

# Exemplo

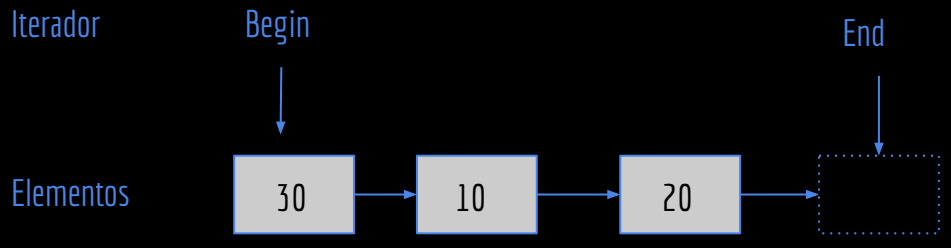



Adaptado de Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.

```cpp
#include<iostream>
#include<list>

int main(){
   std::list<int> lista;
   lista.push_back(10);
   lista.push_back(20);
   lista.push_front(30);

   std::list<int>::iterator it;
   for(it = lista.begin(); it != lista.end(); ++it)
       std::cout << *it << std::endl;

   return 0;
}
```

Repita enquanto o iterador for diferente do final da lista.
A função `end()` retorna um elemento especial que representa o final da lista. Esse não é um elemento válido da lista.

# Exemplo

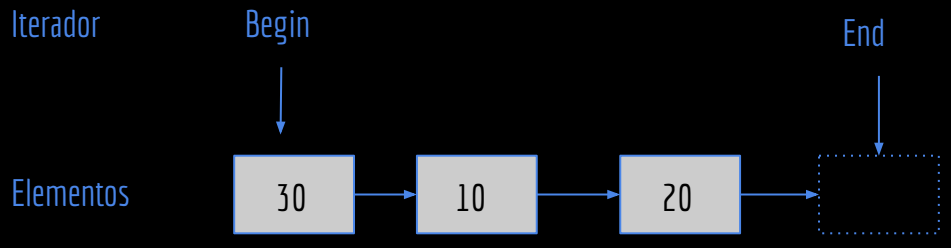



Adaptado de Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.

```
#include<iostream>
#include<list>

int main(){
   std::list<int> lista;
   lista.push_back(10);
   lista.push_back(20);
   lista.push_front(30);

   std::list<int>::iterator it;
   for(it = lista.begin(); it != lista.end(); ++it)
       std::cout << *it << std::endl;

   return 0;
}
```

O operador ++ pode ser usado em um iterador, e faz com que ele passe para o próximo elemento.

# Exemplo




Adaptado de Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.

```
#include<iostream>
#include<list>

int main(){
   std::list<int> lista;
   lista.push_back(10);
   lista.push_back(20);
   lista.push_front(30);

   std::list<int>::iterator it;
   for(it = lista.begin(); it != lista.end(); ++it)
       std::cout << *it << std::endl;

   return 0;
}
```

Um iterador **aponta** para um elemento do container. Se você quer acessar o elemento, precisa usar o operador de derreferencia *, como em um ponteiro convencional.

# Faça você mesmo

Crie uma lista de Ponteiros para Pessoa.

Insira 3 pessoas na lista.

Itere na lista, exibindo o nome de cada pessoa.

# Exemplo

```cpp
#include<iostream>
#include<list>

#include "Pessoa.hpp"

int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111,15});//adicionando no final da lista
   pessoas.push_back(new Pessoa{"Maria", 22222222222,16});//adicionando no final da lista
   Pessoa* p3{new Pessoa{"Pedro", 33333333333,20}};
   pessoas.push_front(p3);//adicionando no início da lista

   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++){//a lista contém ponteiros para pessoas, e não pessoas
       //o itPes é um “ponteiro” que aponta para um ponteiro de pessoa!
       std::cout << (*it)->getNome() << std::endl;
       delete *it;//não esqueça de deletar os ponteiros para desalocar a memória
   }

   return 0;
}
```

# Removendo um item

Para remover um item da lista temos duas opções principais:

`remove`

Recebe o **objeto** que deve ser removido (via referência).

`erase`

Recebe um **iterator que aponta para o objeto** que deve ser removido.

# Exemplos

```cpp
#include<iostream>
#include<list>

#include "Pessoa.hpp"

int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
           break;
   if(it != pessoas.end()){//encontrou ou chegou no final da lista?
       //3 OPÇÕES
   }
   std::cout << "Lista atualizada" << std::endl;
   for(it = pessoas.begin(); it != pessoas.end(); it++){
       std::cout << (*it)->getNome() << std::endl;
       delete *it;
   }
   return 0;
}
```

# Exemplos - Opção 1

```cpp
#include<iostream>
#include<list>

#include "Pessoa.hpp"

int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
           break;
   if(it != pessoas.end()){//encontrou ou chegou no final da lista?
       pessoas.remove(p3);
       delete p3;
   }
   std::cout << "Lista atualizada" << std::endl;
   for(it = pessoas.begin(); it != pessoas.end(); it++){
       std::cout << (*it)->getNome() << std::endl;
       delete *it;
   }
   return 0;
}
```

Nesse caso o loop é irrelevante, e só gastamos processamento.

# Exemplos - Opção 2

```cpp
#include<iostream>
#include<list>

#include "Pessoa.hpp"

int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
           break;
   if(it != pessoas.end()){//encontrou ou chegou no final da lista?
       pessoas.remove(*it);
       delete *it;
   }
   std::cout << "Lista atualizada" << std::endl;
   for(it = pessoas.begin(); it != pessoas.end(); it++){
       std::cout << (*it)->getNome() << std::endl;
       delete *it;
   }
   return 0;
}
```

Usando o valor apontado pelo iterator, que é justamente um ponteiro para Pessoa. **O resultado é o mesmo que o exemplo anterior.**

# Exemplos - Opção 3

```cpp
#include<iostream>
#include<list>

#include "Pessoa.hpp"

int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
           break;
   if(it != pessoas.end()){//encontrou ou chegou no final da lista?
       pessoas.erase(it);
       delete *it;
   }
   std::cout << "Lista atualizada" << std::endl;
   for(it = pessoas.begin(); it != pessoas.end(); it++){
       std::cout << (*it)->getNome() << std::endl;
       delete *it;
   }
   return 0;
}
```

Passamos o iterador para o erase.

# Uma questão de desempenho

Em uma lista encadeada, a função `remove` é **obrigada a visitar todos os itens da lista até encontrar o item que precisa ser removido.**

Qual o problema criado com a construção a seguir?

```
int main(){
    std::list<Pessoa*> pessoas;
    pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
    pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
    Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
    pessoas.push_back(p3);

    std::list<Pessoa*>::iterator it{pessoas.begin()};
    for( ; it != pessoas.end(); it++)
        if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
            break;
    if(it != pessoas.end()){//encontrou ou chegou no final da lista?
        delete *it;
        pessoas.remove(*it);
    }
    //...
    return 0;
}
```

# Uma questão de desempenho

Em uma lista encadeada, a função `remove` é **obrigada a visitar todos os itens da lista até encontrar o item que precisa ser removido.**

Qual o problema criado com a construção a seguir?

```
int main(){
    std::list<Pessoa*> pessoas;
    pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
    pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
    Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
    pessoas.push_back(p3);

    std::list<Pessoa*>::iterator it{pessoas.begin()};
    for( ; it != pessoas.end(); it++)
        if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
            break;
    if(it != pessoas.end()){//encontrou ou chegou no final da lista?
        delete *it;
        pessoas.remove(*it);
    }
    //...
    return 0;
}
```

Percorre uma vez para encontrar o item, e depois percorre mais uma vez para remover.

# Uma questão de desempenho

A função `erase` recebe um iterator que aponta para o objeto a ser removido, e não o objeto em si.

**Um iterator armazena internamente informações de posição.**

**O item pode ser removido diretamente!**

Em uma lista duplamente encadeada, isso é ainda mais eficiente.

A remoção de um item em uma lista duplamente encadeada não exige a realocação dos itens posteriores ao item removido, como seria necessário em um vetor (vector) por exemplo.

# Então...

Como você julga esse trecho?

```
int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   //...
   pessoas.remove(p3);
   delete p3;

   //...
   return 0;
}
```

# Então...

Como você julga esse trecho?

**Está correto!**
Estamos utilizando remove e o objeto, mas não gastamos tempo com um loop para procurar o objeto! O remove se vira para procurar!

```
int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   //...
   pessoas.remove(p3);
   delete p3;

   //...
   return 0;
}
```

# Então...

Como você julga esse trecho?

```
int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   //...
   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
           break;
   if(it != pessoas.end()){//encontrou ou chegou no final da lista?
       pessoas.erase(it);
       delete *it;
   }

   //...
   return 0;
}
```

# Então...

```
int main(){
    std::list<Pessoa*> pessoas;
    pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
    pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
    Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
    pessoas.push_back(p3);

    //...
    std::list<Pessoa*>::iterator it{pessoas.begin()};
    for( ; it != pessoas.end(); it++)
        if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
            break;
    if(it != pessoas.end()){//encontrou ou chegou no final da lista?
        pessoas.erase(it);
        delete *it;
    }

    //...
    return 0;
}
```

Como você julga esse trecho?

**Está correto!**

Nós mesmos procuramos pelo item. E passamos o iterador para o erase, que não vai gastar tempo procurando mais uma vez.

# Então...

Como você julga esse trecho?

```
int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   //...
   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
           break;
   if(it != pessoas.end()){//encontrou ou chegou no final da lista?
       pessoas.remove(p3);
       OU
       pessoas.remove(*it);
       delete *it;
   }

   //...
   return 0;
}
```

# Então...

Como você julga esse trecho?

Isso é ABSURDO!

```
int main(){
   std::list<Pessoa*> pessoas;
   pessoas.push_back(new Pessoa{"Joao", 11111111111, 15});
   pessoas.push_back(new Pessoa{"Maria", 22222222222, 16});
   Pessoa* p3{new Pessoa{"Pedro", 33333333333, 20}};
   pessoas.push_back(p3);

   //...
   std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")//continua até encontrar Pedro
           break;
   if(it != pessoas.end()){//encontrou ou chegou no final da lista?
       pessoas.remove(p3);
       OU
       pessoas.remove(*it);
       delete *it;
   }

   //...
   return 0;
}
```

# Cuidado!

Note que as 3 formas anteriores “funcionam”, mas apenas duas delas fazem sentido.

# Removendo itens no loop

O exemplo anterior remove apenas a primeira referência de Pedro na Lista.

Mas e se a lista conter mais de um Pedro?

**Vamos criar um algoritmo para isso.**

# Removendo itens no loop

Você consegue ver o problema com o algoritmo a seguir?

```
std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")
           pessoas.erase(it);
```

# Removendo itens no loop

Removemos o item apontado pelo iterador.

**Depois tentamos continuar utilizando o iterador** (que foi deletado) no loop. **Falha de segmentação.**

O C++ não vai te avisar sobre esse seu erro, nem em tempo de compilação, nem execução.

As estruturas da STL não implementam mecanismos de segurança, como os Fail-Fast e Fail-Safe implementados no Java, C# e Python.

Esses mecanismos custam tempo de processamento.

**C++ sempre vai deixar você dar um tiro no pé, caso a outra opção seja gastar processamento desnecessariamente.**

```
std::list<Pessoa*>::iterator it{pessoas.begin()};
   for( ; it != pessoas.end(); it++)
       if((*it)->getNome() == "Pedro")
           pessoas.erase(it);
```

# Utilizando o retorno de erase

A função `erase` **retorna um iterador** que aponta para o elemento válido posterior ao removido.

Basta utilizar esse valor retornado para corrigir o problema.

```
std::list<Pessoa*>::iterator it{pessoas.begin()};
  //removendo todas referências a Pedro
  while(it != pessoas.end()){
      if((*it)->getNome() == "Pedro"){
          delete *it;
          it = pessoas.erase(it);//it recebe o próximo item válido da lista
      }else{
          it++;
      }
  }
```

# Atenção

**Não use list para tudo.**

Entenda que cada estrutura de dados tem seus custos e benefícios.

Benefícios de uma lista encadeada?

Malefícios de uma lista encadeada?

# Atenção

**Não use list para tudo.**

Entenda que cada estrutura de dados tem seus custos e benefícios.

Benefícios de uma lista encadeada: É flexível; remover/adicionar itens é computacionalmente barato.

Malefícios de uma lista encadeada: Exige ponteiros extras para cada nodo internamente (+ memória); pode causar invalidação de cache, e é muito custoso acessar um item pelo seu índice (ex.: acesse o item 5).

# Atenção

**Use as estruturas de dados corretas de acordo com o seu problema.**

Problema clássico que sempre encontro em programas Java, C# e Python.

Não é culpa das linguagens (bom, talvez do Python sim).

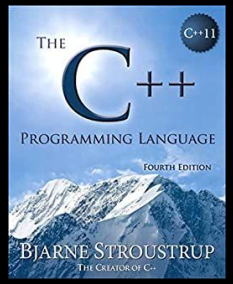

Exemplo: “Programadores” Java gostam de usar ArrayList para tudo, sem sequer parar para pensar se essa estrutura é a melhor para o problema em questão.

# Exercícios

1. Modifique a classe Disciplina:
   a. Em aulas passadas, foi solicitado que fosse utilizado um vetor para representar os alunos que cursam a disciplina. Pesquise sobre os containers da SLT, e substitua esse vetor pelo membro mais adequado da STL (deque, lista, fila, ...?).
   b. Adicione/Modifique as funções membro:
      i. `adicionarAluno(Pessoa* aluno);`
      ii. `removerAluno(Pessoa* aluno);`
      iii. `removerAluno(unsigned long cpf);`
      iv. `tipo_container_stl getAlunos();`

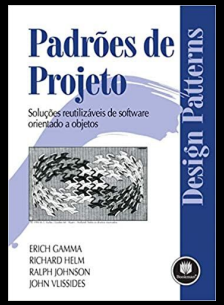
# Referências


Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.

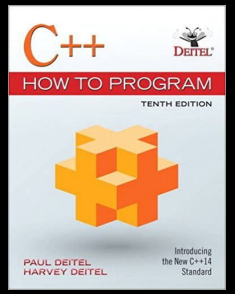


Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.

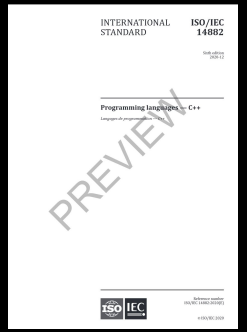


Deitel, H. M., Deitel, P. J. C++: como programar. 5a ed. Pearson Prentice Hall. 2006.



ISO/IEC 14882:2020 Programming languages - C++: www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-6:v1:en

# Licença