

Estruturas de Dados

A. G. Silva, A. von Wangenheim, J. E. Martina

01 de agosto de 2018

- Histórico da linguagem;
- Programa C++: header, source – função `main()`;
- GCC/G++;
- Arquivos objeto, bibliotecas dinâmicas e estáticas;
- `#include`, `#define`, namespaces, `typedefs`;
- Ponteiros, referências, arrays, primitivas, estruturas de controle.

Características do C++

- Linguagem multi-paradigma;
- Programação procedural;
- Orientação a objetos;
- Meta-programação;
- Desalocação memória – decisão cabe ao programador.

- Criada por Bjarne Stroustrup em 1979 no Bell Labs da AT&T e inicialmente chamada de “C with classes”;
- Em 1983 passou a ser chamada C++;
- Ao longo do tempo incorporou novas funcionalidades como:
 - Herança múltipla;
 - Sobreescrita de operadores;
 - Templates;
 - Entre outras coisas que veremos ao longo do curso.

- Possui compatibilidade com C;
- Um compilador C++ deve compilar um programa escrito em C;
- Existem vários compiladores da linguagem;
- É um linguagem padronizada internacionalmente:
 - Em 1998 foi definido um padrão ISO para a linguagem;
 - Revisado em 2003;
 - Revisado em 2011;
 - Revisado em 2014.

- GNU Compiler Collection;
- Compilador originalmente desenvolvido focando a linguagem C, chamava-se GNU C Compiler;
- Ao longo do anos passou a contemplar outras linguagens, como C++.

- Header – definições visíveis a outros headers e ao compilador;
- Sources – corpo de métodos e funções;
- Implemented Headers – headers com implementação que tem que ser instrumentada pelo compilador;
- Função Main – Função inicial invocada pelo executável.

Header de exemplo

```
// in myclass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    void foo();
    float bar;
};

#endif
```


Source de exemplo

```
// in myclass.cpp
#include "myclass.h"
#include <iostream>

void MyClass::foo() {
    std::cout<<"Hello_World!"<<std::endl;
}
```

HPP de exemplo

```
// in myclass.hpp
#include <iostream>

template <class T>
class MyClass {
public:
    void foo() {
        std::cout<<"Hello_World!"<<std::endl;
    }
    T bar;
};
```

Programa de exemplo

```
//in main.cpp
//#include "myclass.h" // defines MyClass
#include "myclass.hpp" // defines MyClass

int main() {
    //MyClass* a = new MyClass();
    MyClass<int>* a = new MyClass<int>();
    a->foo();
    return 0;
}
```

Compilando programa de exemplo

```
$ g++ main.cpp -o hello_world -std=c++11 -Werror  
$ ./hello_world  
$ Hello World!
```

Diretiva `#include`

- Informações sobre arquivos, símbolos e tipos interpretadas no pré-processamento da compilação;
- Avisa ao compilador quais headers ele precisa carregar para pré-processar as instruções do arquivo em questão, mais ou menos como o `import` do Java;
- Pode ser utilizado tanto no header quanto no source;
- Evite `#includes` desnecessários no header, isso torna a etapa de análise mais custosa.

Diretiva #include

- #include pode ser usado para escolher código durante a compilação;

```
#include "structs/my_struct.h"
int main() {
    myStruct ms;
}
```

- \$ g++ include.cc -std=c++11 -Werror

```
#include "my_struct.h"
int main() {
    myStruct ms;
}
```

- \$ g++ include.cc -I structs -std=c++11 -Werror

Diretiva #include

```
#include <list>
```

- Procura o header nos diretórios do sistema e aqueles passados como parâmetro ao compilador.

```
#include "my_struct.h"
```

- Procura o header primeiro no diretório que contém o arquivo sendo compilado, e depois nos diretórios do sistema e aqueles passados como parâmetro ao compilador.

Diretiva #define

- Atribui um valor a um símbolo;
- Cada vez que o pré-processador encontra este símbolo no código ele substitui o símbolo por esse valor.

```
#define max_iterations 1000
```

- Em C++11, para a definição de constantes devemos usar *const* pois este é *type-safe* e são tratadas quase como #define pelo compilador.

Diretiva #define

- Também pode ser usado para definir “macros”, pequenos trechos de código que são colados cada vez que o pré-processador identifica o símbolo associado.

```
#define getmax(a,b) ((a)>(b)?(a):(b))

int main() {
    int x = 10;
    int y = getmax(x,5); // y vale 10
}
```

- Com a substituição feita ele fica assim:

```
#define getmax(a,b) ((a)>(b)?(a):(b))

int main() {
    int x = 10;
    int y = ((x)>(5)?(x):(5)); // y vale 10
}
```

- Só devemos usar quando o código é pequeno e existe um real conhecimento do fluxo das instruções;
- Problemas podem acontecer e o código colado pelo pré-processador pode tomar um rumo inesperado;
- Como o código é “colado”, não existe ônus de ter que colocar o endereço da instrução na pilha como em uma chamada de função;
- Boas práticas em C++11 dizem que você só deve usar defines para fazer “Head Guarding”

- Estabelecem o domínio ao qual declarações (classes, structs, metaclasses) fazem parte;
- É utilizado para organizar o código em diferentes domínios, que dependendo da estrutura de diretórios do código fonte;
- Apresentam uma leve semelhança com a organização em pacotes do java;
- Também serve para eliminar ambiguidades.

Diretiva namespace

```
#include "myclass.h"
#include <iostream>
void MyClass::foo() {
    std::cout<<"Hello_World!"<<std::endl;
}
```

- Passamos a não precisar chamar as funções com seu nome completo

```
#include "myclass.h"
#include <iostream>

using namespace std;
void MyClass::foo() {
    cout<<"Hello_World!"<<endl;
}
```

HPP Exemplo namespace

```
// in myclass.hpp
#include <iostream>

using namespace templates;

template <class T>
class MyClass {
public:
    void foo() {
        std::cout<<"Hello_World!"<<std::endl;
    }
    T bar;
};
```

Programa Exemplo namespace

```
// in main.cpp
#include "myclass.h"    // defines MyClass
#include "myclass.hpp"  // defines MyClass

using namespace std;

int main() {
    MyClass* a; // ambiguidade
    MyClass<int>* b; // ambiguidade
}
```

- Qual das implementações de MyClass foi usada?

Programa Exemplo namespace

```
// in main.cpp
#include "myclass.h" // defines MyClass
#include "myclass.hpp" // defines MyClass

using namespace std;

int main() {
    MyClass* a;
    templates::MyClass<int>* b;
}
```

- Se resolve dando o nome completo!

Diretiva typedef

```
// in main.cpp
#include "myclass.h"    // defines MyClass
#include "myclass.hpp"  // defines MyClass

using namespace std;
typedef MyClass<int> MinhaClasseInteira;

int main()
{
    MyClass* a;
    MinhaClasseInteira* b;
}
```

- Se resolve criando uma nova definição de tipo.

Diretiva using

```
// in main.cpp
#include "myclass.h"    // defines MyClass
#include "myclass.hpp"  // defines MyClass

using namespace std;
using MinhaClasseInteira = MyClass<int>;

int main() {
    MyClass* a;
    MinhaClasseInteira* b;
}
```

- Recomendação de uso em C++11;

Diretiva using

```
// in main.cpp
#include "myclass.h" // defines MyClass
#include "myclass.hpp" // defines MyClass

using namespace std;
template <typename T>
using MinhaClasseNaoAmbigua = templates::MyClass<int>;

int main() {
    MyClass* a;
    MinhaClasseNaoAmbigua<int>* b;
}
```

- Recomendação de uso em C++11;
- Consigo resolver ambiguidade e manter templates.

Tipos primitivos

- O tamanho é medido em word ou frações de words;
- O tamanho é flexível e definido em cada plataforma;
- Inteiros com e sem sinal:
 - char, unsigned char
 - short, unsigned short
 - int, unsigned int
 - long, unsigned long
 - long long, unsigned long long

Tipos primitivos

- Ponto flutuante:
 - float
 - double
 - long double
- Outros tipos:
 - bool
 - void
 - decltype(nullptr)

Variáveis, ponteiros e referências

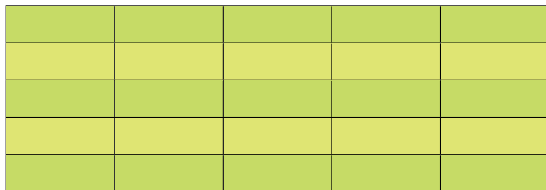
- Variável quando declarada tem um tipo que determina o seu tamanho;
- Variável faz com que o S.O. reserve memória suficiente para o tipo e disponibilize para o programa;
- Ponteiro é uma variável especial que guarda endereços e não dados;
- Ponteiro quando declarado tem um tipo que determina qual o tipo do endereço que ele armazena;
- Ponteiro faz com que o S.O. reserve memória suficiente para armazenar um endereço de memória;
- Referência é um segundo nome para uma variável.

Variáveis, ponteiros e referências

- * é chamado de indireção. Refere-se ao valor do endereço de memória guardado no ponteiro;
- & retorna o endereço de memória de uma variável. Também serve para criar uma referência;
- Podemos ter uma dupla indireção com **;
- Podemos (em C++11) ter uma referência não constante com &&.

Variáveis, ponteiros e referências

- Modelo da memória de um computador:



- É uma sequência de palavras de memória endereçáveis pela sua posição.

Modelo de memória

- `int a = 10;`
- Política do S.O. Escolhe o endereço 6;
- Modelo da memória de um computador:

10				

Declarando variáveis e ponteiros

- `int a = 10;`
- `int* b = &a;`
- Política do S.O. Escolhe o endereço 9 para o ponteiro;
- Modelo da memória de um computador:

10			6	

Declarando variáveis e ponteiros

```
#include <iostream>

using namespace std;

int main() {
    int a = 5;
    int * b = &a;
    cout<<"O valor de a" <<a<<" o endereço de a é" <<&
        a<<endl;
    cout<<"b contém o endereço" <<b<<" o conteúdo do
        endereço de b é:" <<*b<<endl ;
}
```

- Executar algumas vezes no VPL.

Declarando variáveis e ponteiros

```
#include <iostream>

using namespace std;

int main(){
    int a = 5;
    int * b = &a;
    *b += a;
    a *= (*b);
    std::cout<<a<<std::endl;
}
```

- O que imprime esse código?

Referências

- Uma referência é como um “apelido” a uma outra variável;
- O S.O. não aloca uma nova word para uma referência;
- Tal qual um ponteiro, a referência apenas “referencia” algum endereço de memória;
- Uma vez criada, não há como distinguir entre uma referência e o dado por ela referenciado;
- Referência & só aponta para um valor durante a sua vida;
- Referência && pode mudar (somente no std11);

Variáveis, ponteiros e referências

```
int main() {  
    int a = 3;  
    int b = 4;  
    int * p = &b;  
    int & r = b;  
    r += 2; // b agora vale 6  
    p = &a; // p aponta para "a"  
    r = a; // r ainda referencia b.  
}
```

- Quanto vale b?

Arrays

```
int main() {  
    int a[5];  
    a[0] = 0;  
    a[4] = -3;  
}
```

- Vetor com alocação estática;
- O vetor inicia no endereço 7 e usa 5 variáveis inteiras;

	0			
-3				

Arrays

```
int main() {  
    int a = 5;  
    int * b = new int[a];  
}
```

- Vetor com alocação dinâmica;
- O vetor inicia no endereço 7 e usa a variáveis inteiras;

		5		

Iterando arrays

```
int main() {  
    int a = 5;  
    int * b = new int[a];  
    int * it = &b[0];  
    (*it++) = 8;  
}
```

- Quais operações são executadas na última linha?

Iterando arrays

```
int main() {  
    int a = 5;  
    int * b = new int[a];  
    int * it = &b[0];  
    (*it++) = 8;  
}
```

- Quais operações são executadas na última linha?
- $(*it++) = 8 \rightarrow$ atribui o valor 8 à posição atual do iterador e incrementa o iterador, fazendo-o apontar para a próxima posição;

- If-Then-Else:

```
if ( expressao booleana ) {  
    //se a expressao e verdadeira executa este  
    bloco  
} else {  
    //se a expressao e falsa executa este bloco  
}
```

Statements

- While:

```
while ( expressao booleana ){  
    // enquanto a expressao for verdadeira executa  
    este bloco  
}
```

- Do-While:

```
do {  
    // executa este bloco enquanto a expressao for  
    verdadeira  
} while( expressao booleana );
```

- For:

```
for ( inicializacao de variaveis ; expressao ;  
      operacao executada a cada loop ) {  
    // Para cada vez que a expressao for  
    verdadeira, executa esse bloco. Se a  
    expressao      for falsa, sai do bloco  
    .  
}
```

- Em algumas versões antigas do standard tem que instanciar a variável antes do loop.

- Enum:

```
enum Paises {  
    Argentina,  
    Brasil,  
    China,  
    Estados Unidos,  
    Inglaterra  
}
```

```
enum Paises {  
    Argentina=9,  
    Brasil=6,  
    China=5,  
    Estados Unidos=13,  
    Inglaterra=10  
}
```

Switch-case

- Seleção Switch-case:

```
int a;  
switch (a) {  
    case:1  
        // se a vale 1, executa este bloco  
        break;  
    case:2  
        // se a vale 2, executa este bloco  
        break;  
    default:  
        // se a vale qualquer outro inteiro, executa  
        este bloco  
}
```

Perguntas?





Este trabalho está licenciado sob uma Licença Creative Commons Atribuição 4.0 Internacional. Para ver uma cópia desta licença, visite

<http://creativecommons.org/licenses/by/4.0/>.

