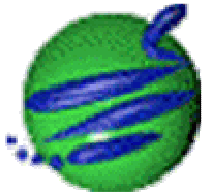
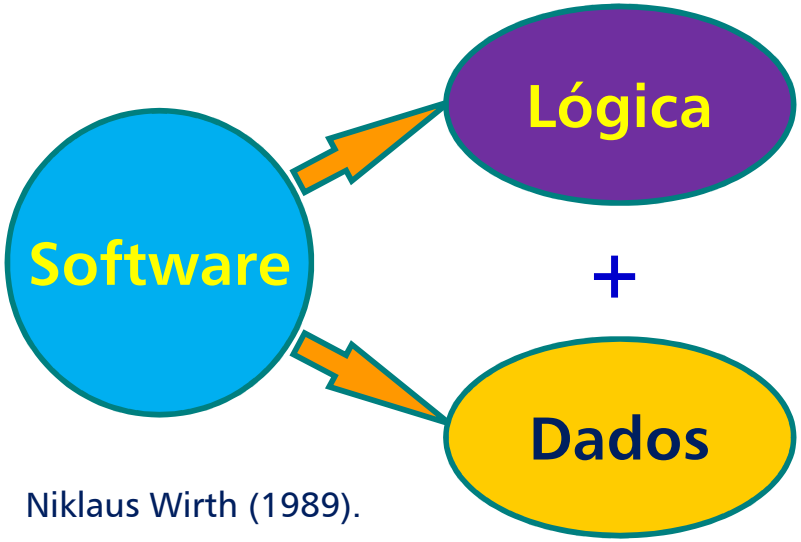


Introdução à Disciplina de Estruturas de Dados

Adair Santa Catarina
Curso de Ciência da Computação
Unioeste – Campus de Cascavel – PR

Fev/2019

Software = Lógica + Dados



Niklaus Wirth (1989).

Estuda como as operações são encadeadas para se chegar no resultado esperado → Algoritmos.

São os elementos manipulados no software → Estruturas de Dados.

Objetivo principal da disciplina Estruturas de Dados: Estudar como os dados são estruturados, armazenados e, principalmente, como podem ser manipulados.

Tipos Primitivos de Dados (GCC-C++11)

Tipo	Min	Max	Bytes (x32)	Bytes (x64)
bool	false (0)	true (1)	1	1
char	-128	127	1	1
unsigned char	0	255	1	1
wchar_t	0	65535	2	2
short	-32768	32767	2	2
unsigned short	0	65535	2	2
int = long	-2147483648	2147483647	4	4
unsigned int = unsigned long	0	4294967295	4	4
long long	-9223372036854775808	9223372036854775807	8	8
unsigned long long	0	18446744073709551615	8	8
float	±1,17549e-38	±3,40282e+38	4	4
double	±2,22507e-308	±1,79769e+308	8	8
long double	±3,3621e-4932	±1,18973e+4932	12	16

O Tipo *void*

O tipo *void* significa que não tem valor definido ou o valor está ausente. É usado em 3 situações:

1. Quando uma função não tem valor a retornar ela retorna *void*. Por exemplo, void exit(int status);

2. Quando a função não tem parâmetros, sendo invocada sem argumentos. Por exemplo, int rand(void);

3. Quando alocamos memória com um ponteiro genérico que, posteriormente será convertido para um tipo definido (*cast*). Por exemplo, void *malloc(size_t size);

Ponteiros (*)

Ponteiros são variáveis que armazenam um endereço de memória. São identificados pelo símbolo *, que precede o identificador da variável.

Um ponteiro pode apontar para o nada (*nullptr*) ou para um tipo qualquer (primitivo ou não), quando houver a alocação de memória.

```
#include <iostream>
using namespace std;
int main(){
   short int *p  = nullptr;
   cout << "p = " << p << endl;
   p = new short int(5);
   cout << "p = " << p << ", *p = " << *p << endl;
   cout << " tam p = " << sizeof(p) << ", tam *p = " << sizeof(*p) << endl;
   delete p;
}
```

Saídas:

```
p = 0;
p = 0x6bae70, *p = 5
tam p = 4, tam *p = 2
```

Referências (&)

Referências são similares aos ponteiros, entretanto elas não podem assumir o valor *nullptr*, pois devem apontar para um elemento já existente.

Por isso uma referência deve ser inicializada em sua declaração. O tipo de uma referência não pode ser alterado e não é conveniente modificá-la.

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

int main(){
   short int p  = 5;
   short int &q = p;
   cout << "q = " << &q << ", q = " << q << endl;
   cout << "tam &q = " << sizeof(&q) << ", tam q = " << sizeof(q) << endl;
}
```

Saídas:
&q = 0x28ff0a, q = 5
tam &q = 4, tam q = 2

Ponteiros (*) e Referências (&)

Ponteiros e Referências também são tipos primitivos da linguagem C++.

Os conteúdos das variáveis destes dois tipos primitivos será sempre um endereço de memória.

O espaço ocupado em memória por ponteiros e referências depende da arquitetura para a qual foi compilada a aplicação.

Em arquiteturas 32 bits, ponteiros e referências ocupam 4 bytes na memória, mesmo que estejam apontando para um *char* ou *long double*. Em 64 bits ponteiros e referências ocupam 8 bytes na memória.

Constantes

Quando é associado um valor a um identificador e este valor não pode ser mais alterado.

Duas maneiras de se definir constantes em C++:
#define e const

```
#include <iostream>
using namespace std;
#define PI 3.1416926539

int main(){
   const char NewLine = '\n';
   float area, raio;

   raio = 3.2
   area = PI * raio * raio;
   cout << "Area = " << area << NewLine;
}
```

Construção de Tipos: Enumerados e Derivados

Além dos tipos primitivos existem tipos que podem ser definidos ou construídos, chamados de enumerados e derivados (vetores, strings, matrizes e registros).

Tipos enumerados permitem criar variáveis que assumam valores específicos, pré-determinados.

```
#include <iostream>
using namespace std;
typedef enum Color {red, green, blue, cyan = 10, magenta, yellow} TColor;

int main(){
   TColor a, b, c;
   a = red; b = blue; c = yellow;
   cout << "red a = " << a << endl
        << "blue b = " << b << endl
        << "yellow c = " << c << endl;
 }
```

Saídas:
red a = 0
blue b = 2
yellow c = 12

Vetores

Vetores são estruturas homogêneas unidimensionais que podem conter diversos elementos.

Homogêneas porque seus elementos são todos de um mesmo tipo. E unidimensionais porque os elementos são acessados por um índice único.

```
#include <iostream>
using namespace std;
int main(){
   int pares[5];
   int *impares = new int[5];
   for (int i = 0; i < 10; ++i){
      if (i%2 == 0) pares[i /2] = i;
      else impares[i/2] = i;
   }
   for (int i = 0; i < 5; ++i) cout << pares[i] << " " << impares[i] << " ";
   delete[] impares;
}
```

Matrizes

Matrizes são estruturas similares aos vetores. Nas matrizes são necessários dois ou mais índices para individualizar seus elementos.

```
#include <iostream>
using namespace std;
int main(){
   int matrizA[3][4], matrizB[3][4];
   int **matrizC;
   matrizC = new int*[3];
   for (int i = 0; i < 3; ++i) matrizC[i] = new int[4];
   for (int i = 0; i < 3; ++i){
      for (int j = 0; j < 4; ++j) {
         matrizA[i][j] = 2*i-j;
         matrizB[i][j] = i + 5*j;
         matrizC[i][j] = matrizA[i][j] + matrizB[i][j];
      }
   }
   for (int i = 0; i < 3; ++i) delete[] matrizC[i];
   delete[] matrizC;
}
```

Registros

Registros são estruturas agregadas heterogêneas. Ou seja, podem conter elementos de tipos distintos.

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

typedef struct SAluno{
   string nome;
   int idade;
   char serie;
} TAluno;

int main(){
   TAluno Cal;
   TAluno *Vet = new TAluno;
   Cal.nome = "Joaquim Jose da Silva Xavier"; Cal.idade = 18; Cal.serie = '1';
   Vet->nome = "Luis Alves de Lima e Silva"; Vet->idade = 21; Vet->serie = '3';
   cout << Cal.nome << endl << Cal.idade << endl << Cal.serie << endl  << endl;
   cout << Vet->nome << endl << Vet->idade << endl << Vet->serie << endl;
   delete Vet;
}
```

Funções

São grupos de comandos que, em conjunto, realizam uma tarefa. Elas podem ou não retornar algum valor após sua execução.

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

int max (int a, int b){ return((a >= b) ? a: b); }

void printLine(int size = 80){ //parâmetro default
   for (int i = 0; i < size; ++i) cout << '-';
   if ((size%80) != 0) cout << endl;
}

int main(){
   int x  = 4, y = 6, m = max(x, y);
   printLine(20);
   cout << "x = " << x << "\ty = " << y << endl;
   printLine(40);
   cout << "Maximo = " << m << endl;
   printLine();
}
```

Escopo de Variáveis

Escopo corresponde ao espaço de existência de uma variável. Ou seja, em que porção do programa ela é reconhecida.

No C++ variáveis declaradas no início do programa, fora do corpo da primeira função, possuem escopo global. Ou seja, existem e são acessíveis em qualquer local do programa.

Já os parâmetros de uma função, variáveis declaradas dentro de uma função ou bloco ({ }) possuem escopo local. Ou seja, existem e são acessíveis apenas dentro da função ou do bloco onde foram declaradas.

Escopo de Variáveis

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

double x; //variável global
unsigned int fatorial (unsigned int a){
   unsigned int fat = 1; //parâmetro a e fat são variáveis locais
   if (a < 2) return(fat);
   else while (a > 1){
      fat = fat * a;
      --a;
   }
   return(fat);
}
int main(){
   int number; //variável local
   cout << "Qual o numero [0..12] ?";
   cin >> number;
   x = fatorial(number);
   cout <<fixed<<setprecision(0)<< "Fatorial (" << number << ") = " <<x < endl;
   cout << setprecision(6) << defaultfloat;
}
```

Passagem de Parâmetros

Em funções que utilizam argumentos precisamos declarar variáveis, chamadas parâmetros, para receber os respectivos argumentos.

Os parâmetros da função são variáveis locais. Ou seja, são alocadas quando se inicia a execução da função e desalocadas no término da mesma.

No C++, quando chamamos uma função os argumentos podem ser passados aos parâmetros de 3 formas distintas: por valor, por ponteiros e por referência.

Passagem de Parâmetros por Valor

Os valores dos argumentos são copiados para os parâmetros da função. Alterações nos parâmetros não afetam os argumentos.

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

void Swap(int a, int b){//os parâmetros a e b são cópias dos argumentos
   int temp = a;
   a = b;
   b = temp;
   cout << "Durante --> a = " << a << ", b = " << b << endl;
}

int main(){
   int x = 5, y = 9;
   cout << "Antes --> x = " << x << ", y = " << y << endl;
   Swap(x, y);
   cout << "Depois --> x = " << x << ", y = " << y << endl;
}
```

Saídas:
Antes --> x = 5, y = 9
Durante --> a = 9, b = 5
Depois --> x = 5, y = 9

Passagem de Parâmetros por Ponteiro

O endereço do argumento é passado para o ponteiro parâmetro. Alterações nos parâmetros modificam os argumentos.

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

void Swap(int *a, int *b){// os ponteiros a e b copiam os endereços dos argumentos
   int temp = *a;
   *a = *b;
   *b = temp;
   cout << "Durante --> a = " << *a << ", b = " << *b << endl;
}

int main(){
   int x = 5, y = 9;
   cout << "Antes --> x = " << x << ", y = " << y << endl;
   Swap(&x, &y); // os endereços dos argumentos x e y são enviados aos parâmetros
   cout << "Depois --> x = " << x << ", y = " << y << endl;
}
```

Saídas:
Antes --> x = 5, y = 9
Durante --> a = 9, b = 5
Depois --> x = 9, y = 5

Passagem de Parâmetros por Ponteiro

```
#include <iostream>
#include <cstring>
using namespace std;
void Reverse(char *s){ // o vetor de char (string) foi recebido como ponteiro
   int i, j = strlen(s);
   char c;
   for (i = 0; i < j/2; ++i){
      c = s[i];
      s[i] = s[j - i - 1]; //está alterando o vetor argumento
      s[j - i - 1] = c; //está alterando o vetor argumento
   }
}
int main(){
   char str[30] = "Arara eh troll!";
   cout << str << endl;
   // o ponteiro base do vetor/matriz é o argumento enviado ao parâmetro
   Reverse(str);
   cout << str << endl;
}
```

Passagem de Parâmetros por Referência

Uma referência ao argumento é copiada no parâmetro. Dentro da função a referência permite acessar o argumento, possibilitando sua modificação.

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

void Swap(int &a, int &b){//a e b são referências às variáveis parâmetro x e y
   int temp = a;
   a = b;
   b = temp;
   cout << "Durante --> a = " << a << ", b = " << b << endl;
}

int main(){
   int x = 5, y = 9;
   cout << "Antes --> x = " << x << ", y = " << y << endl;
   Swap(x, y); // os argumentos x e y são enviados aos parâmetros
   cout << "Depois --> x = " << x << ", y = " << y << endl;
}
```

Saídas:
Antes --> x = 5, y = 9
Durante --> a = 9, b = 5
Depois --> x = 9, y = 5