PEF – 5743 – Computação Gráfica Aplicada à Engenharia de Estruturas

Prof. Dr. Rodrigo Provasi

e-mail: provasi@usp.br

Sala 09 – LEM – Prédio de Engenharia Civil

Introdução ao C++

- C++ é uma linguagem *middle level (*como Java e C)
- C++ é uma linguagem estruturadas
- C++ é uma linguagem para programadores (não é como algumas linguagens, um *script*)

Palavras chave (*Keywords*) no C++

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Características

- C++ utiliza diversos arquivos para representar o código.
- Isso permite que eles sejam compilados independentemente.
- Recompilar o código → recompilar os arquivos alterados.
- Necessário *linkar* os arquivos depois!

Tipos Básicos

- C possui 5 tipos básicos: *char*, *int*, *float*, *double* e *void*.
- C++ adiciona ainda *bool* e *wchar_t*. (discutidos mais a frente).
- Todos os tipos derivam desses tipos básicos.

Type	Typical Size in Bits	Minimal Range
char	8	–127 to 127
unsigned char	8	0 to 255
signed char	8	–127 to 127
int	16 or 32	–32,767 to 32,767
unsigned int	16 or 32	0 to 65,535
signed int	16 or 32	same as **int**
short int	16	–32,767 to 32,767
unsigned short int	16	0 to 65,535
signed short int	16	same as **short int**
long int	32	–2,147,483,647 to 2,147,483,647
signed long int	32	same as **long int**
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	80	Ten digits of precision

Nomeando variáveis

- Em C++ existem algumas regras para nomear-se as variáveis.
- Elas devem iniciar em letras ou sublinhado e seguir com letras, números ou sublinhado.

Correct	Incorrect
Count	1count
test23	hi!there
high_balance	high...balance

Nomeando variáveis

- Para criar uma variável usa-se a seguinte sintaxe:

type variable_list;

```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Escopo das variáveis

- A variável 'x' nas funções ao lado não são a mesma variável.
- Como estão declaradas em um bloco de código, elas valem apenas naquele bloco.

```
void func1(void)
{
  int x;

  x = 10;
}

void func2(void)
{
  int x;

  x = -199;
}
```

Declaração em C *versus* C++

- No C, precisa-se declarar todas as variáveis no início.
- No C++, é possível declarar as variáveis somente quando elas serão usadas.

```
/* This function is in error if compiled as
   a C program, but perfectly acceptable if
   compiled as a C++ program.
*/
void f(void)
{
  int i;

  i = 10;

  int j;  /* this line will cause an error */

  j = 20;
}
```

Parâmetros de função

- Uma função usa parâmetros e seus tipos precisam ser explicitados.

```
/* Return 1 if c is part of string s; 0 otherwise */
int is_in(char *s, char c)
{
  while(*s)
    if(*s==c) return 1;
    else s++;

  return 0;
}
```

Variáveis Globais

- No C++ é possível definir uma variável global
- No caso *count* é global e pode ser acessada por qualquer função!
- *func2* declara um *count* que sobrescreve o global.

```
#include <stdio.h>
int count;  /* count is global */

void func1(void);

void func2(void);

int main(void)
{
  count = 100;
  func1();

  return 0;
}

void func1(void)
{
  int temp;

  temp = count;
  func2();
  printf("count is %d", count); /* will print 100 */
}

void func2(void)
{
  int count;

  for(count=1; count<10; count++)
    putchar('.');
}
```

Modificadores

- *const*: define uma variável com valor fixo que não pode ser alterado (apesar de ser possível fornecer um valor inicial).
- *extern*: permite a importação de funções de um arquivo externo.
- *static:* cria uma variável que funciona como global no escopo em que ela foi criada. Por exemplo, se ela pertence à uma classe, todas as classes veem essa variável como global (não muda entre os vários objetos dessa classe).

extern
static
register
auto

Inicializando uma variável

- Para inicializar uma variável, declara-se seu tipo, um nome (identificador) e atribui-se um valor com o operador de atribuição (=).

type variable_name = value;

Some examples are

```
char ch = 'a';
int first = 0;
float balance = 123.23;
```

Comandos em *strings*

- *Strings* são um 'vetor' de caracteres e armazenam palavras e/ou frases.
- É possível incorporar alguns comandos usando a barra invertida em *strings*.

Code	Meaning
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\"	Double quote
\'	Single quote
\0	Null
\\	Backslash
\v	Vertical tab
\a	Alert
\?	Question mark
\N	Octal constant (where N is an octal constant)
\xN	Hexadecimal constant (where N is a hexadecimal constant)

Operadores

- Existem diversos operadores no C++:
  - = : operador de atribuição. Serve para definir valor às variáveis.
  - ++, -- : operadores de incremento / decremento. Acrescentam ou subtraem 1 da variável.
  - + , - , * , / : operadores matemáticos. Faz operações entre os tipos.
  - % : operador *modulus*. Resto da divisão entre dois inteiros.

Operadores

Operator	Action
–	Subtraction, also unary minus
+	Addition
*	Multiplication
/	Division
%	Modulus
– –	Decrement
++	Increment

Precedência dos operadores:

highest	++ – –
	– (unary minus)
	* / %
lowest	+ –

Operadores

Relational Operators

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
= =	Equal
!=	Not equal

Logical Operators

Operator	Action
&&	AND
||	OR
!	NOT

Operadores

- Operador ? → Equivale a um *if*. Sintaxe: *exp ? expForTrue : expForFalse.*

- *Exemplo:*

```
x = 10;
y = x > 9 ? 100 : 200;
```

equivalente →

```
x = 10;
if(x > 9) y = 100;
else      y = 200;
```

Operadores

- Operadores * e &: são para controle e acesso de memória.
  - & → retorna o endereço de memória da variável.
  - * → retorna o valor da variável naquele endereço de memória.
- Se *count = 200*, *m* contém o endereço de *count* e q recebe 200.
- Para declarar a variável m como ponteiro, tem-se:

  *int *m, count;*

```
m = &count;

q = *m;
```

Operadores

- Operador ponto (.) e seta (->): permite acessar elementos individuais de estruturas e classes.

```
struct employee
{
  char name[80];
  int age;
  float wage;
} emp;

struct employee *p = &emp; /* address of emp into p */
```

you would write the following code to assign the value 123.23 to the **wage** member of structure variable **emp**:

```
emp.wage = 123.23;
```

However, the same assignment using a pointer to **emp** would be

```
p->wage = 123.23;
```

Operadores

- Operador (): altera a ordem de precedência.
- Operador []: permite o acesso à itens de um vetor (discutidos depois).

Booleanos

- No C++ existe o *boolean*, um tipo que permite atribuir verdadeiro (*true*) ou falso (*false*) à variável.
- Usado para verificações em instruções decisórias (como *if*) e loops (como *while* e *for*).

Intrução *if* e *if … else*

```
/* Magic number program #2. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Right **");
  else printf("Wrong");

  return 0;
}
```

```
if(i)
{
  if(j) statement 1;
  if(k) statement 2; /* this if */
  else  statement 3; /* is associated with this else */
}
else statement 4; /* associated with if(i) */
```

```
if (expression) statement;
else
  if (expression) statement;
  else
    if (expression) statement;
    .
    .
    .
    else statement;
```

Intrução *switch*

- Um *switch* funciona para casos de inteiros ou caracteres constantes.
- Estrutura básica:

```
switch (expression) {
  case constant1:
    statement sequence
    break;
  case constant2:
    statement sequence
    break;
  case constant3:
    statement sequence
    break;
  .
  .
  .
  default
    statement sequence
}
```

Loops

- Comando *for* → for(*initialization*; *condition*; *increment*) *statement*;
- Comando *while* → while(*condition*) *statement*;
- Comando *do…while* → do{ *statement*; } while(*condition*);

- Diferenças:
  - *for* inicialização e incrementos na estrutura do comando.
  - *for* e *while* não são executados se *condition* é false na primeira verificação. Já o *do… while* só verifica após a primeira execução.
  - *while* e *do … while* precisam ter inicializações e incrementos feitos antes / dentro do *loop*.

Comandos de mudança de fluxo

- Permitem criar 'saltos' no programa:
  - *return :* usado para o retorno de função.
  - *goto :* vai para um determinado *label*.
  - *break :* encerra o *loop* em execução.
  - *continue:* força a próxima iteração do *loop* em execução, sem executar os comandos seguintes no *loop*.

```
x = 1;
loop1:
  x++;
  if(x<100) goto loop1;
```

Array

- Um vetor (ou *array*) é uma coleção de mesmos tipos que pode ser acessada sequencialmente.

- Sua declaração é:

type var_name[size];

```
double balance[100];
```

Array

- O acesso se dá pelo operador [] e cada posição do vetor corresponde a um valor alocado em uma posição de memória.

- Para uma variável (memória começando de 1000):

  char a[7];

Element	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Address	1000	1001	1002	1003	1004	1005	1006

Ponteiro para um vetor

- Pode-se criar um ponteiro para apontar para o vetor da seguinte forma:

```
int *p;
int sample[10];
p = sample;
```

- Nesse caso se atribuísse-se à p &sample[o], ter-se-ia o mesmo efeito.

Array

- Um *array* tem tamanho fixo sendo igual ao número de entradas vezes o espaço ocupado por aquele tipo de variável.
- Não há um controle das posições em C/C++ sendo que o código a seguir não gera exceções:

```
int count[10], i;
/* this causes count to be overrun */
for(i=0; i<100; i++) count[i] = i;
```

Array

- É possível passar um *array* como parâmetro. Existem algumas formas:

```
void func1(int *x) { … }/* pointer */
void func1(int x[10]) { … } /* sized array */
void func1(int x[]) { … }/* unsized array */
```

Arrays bidimensionais

- No C/C++ é possível criar *arrays* bidimensionais.

```
int d[10][20];
```

- E seu acesso fica:

```
d[1][2]
```

Arrays

- O que foi visto pode ser expandido para *n* dimensões, ou seja, pode-se criar *arrays* multidimensionais.

Inicialização

- Para se inicializar um *array* usamos a seguinte sintaxe:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int sqrs[10][2] = {1, 1, 2, 4, 3, 9, 4, 16, 5, 25, 6, 36, 7, 49, 8, 64,
9, 81,
10, 100
};
```

Inicialização

- O segundo exemplo pode ser inicializado com colchetes:

```
int sqrs[10][2] = {{1, 1},{2, 4},{3, 9}, {4, 16},
{5, 25}, {6, 36},{7, 49}, {8, 64}, {9, 81}, {10, 100}};
```

Funções

- A forma geral de uma função é:

ret-type function-name(*parameter list*)

{

    'body of the function

}

Funções

- Para um valor ser modificado na função, deve-se passar por referência:

```
void swap(int *x, int *y)
{
    int temp;
    temp = *x; /* save the value at address x */
    *x = *y; /* put y into x */
    *y = temp; /* put x into y */
}
```

Funções

- Na chamada fica:

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

int main(void)
{
    int i, j;
    i = 10;j = 20;
    swap(&i, &j); /* pass the addresses of i and j */
    return 0;
}
```

Funções

- Chamada com um *array* de parâmetro:

```
#include <stdio.h>
#include <ctype.h>

void print_upper(char *string);
int main(void)
{
     char s[80];
     gets(s);
     print_upper(s);
     printf("\ns is now uppercase: %s", s);
     return 0;
}
```

```
/* Print a string in uppercase. */
void print_upper(char *string)
{
     register int t;
     for(t=0; string[t]; ++t)
     {
          string[t] = toupper(string[t]);
          putchar(string[t]);
     }
}
```

Funções

- É possível retornar um ponteiro em uma função:

```
/* Return pointer of first occurrence of c in s. */
char *match(char c, char *s)
{
   while(c!=*s && *s) s++;
   return(s);
}
```

Funções

- Sua chamada fica:

```
#include <stdio.h>

char *match(char c, char *s); /* prototype */

int main(void)
{
     char s[80], *p, ch;
     gets(s);
     ch = getchar();
     p = match(ch, s);
     if(*p) /* there is a match */
     printf("%s ", p);
     else
     printf("No match found.");
     return 0;
}
```

Funções

- Como pode ser visto nos exemplos anteriores, todas as funções devem ter um protótipo declarado antes de serem usadas.
- A declaração é:

type func_name(type parm_name1, type parm_name2,. . ., type parm_nameN);

Estruturas (*structs*)

- Uma estrutura é uma coleção de variáveis referenciavas por um único nome.
- Permite o acesso e agrupamento dessas variáveis de forma simples.

```
struct addr
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
};
```

structs

- Para declarar uma *struct:*

```
addr addr_info;
```

- No C seria:

```
struct addr addr_info;
```

struct

- O acesso aos dados de uma estrutura é feito pelo operador . :

```
addr_info.zip = 12345;
```

- É possível declarar um ponteiro de uma estrutura:

```
struct addr *addr_pointer;
```

struct

- Assim, pode-se fazer:

```
struct bal {
  float balance;
  char name[80];
} person;

struct bal *p;  /* declare a structure pointer */

p = &person;
```

struct

- E o acesso as variáveis da estrutura se dá pelo operador -> :

```
p->balance
```

Enumerations

- *Enumerations* são uma coleção de número inteiros com nomes e que podem ser usados no código.
- Servem para melhor representar certos conjuntos finitos de dados.

- Exemplo:

```
enum coin { penny, nickel, dime, quarter,
            half_dollar, dollar};
```

Enumerations

- Um possível uso pode ser:

```
money = dime;
if(money==quarter) printf("Money is a quarter.\n");
```

Enumerations

- As *enumerations* seguem a numeração do zero ao número máximo de 1 em 1.
- Valores podem ser atribuídos as *enumerations* que podem ser usados em operações.

```
enum coin { penny, nickel, dime, quarter=100,
            half_dollar, dollar};
```

Enumerations

- Agora os valores ficam:

penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

Enumerations

- Um bom uso de *enumerations* é em um *switch*:

```
switch(money) {
    case penny: printf("penny");
        break;
    case nickel: printf("nickel");
        break;
    case dime: printf("dime");
        break;
    case quarter: printf("quarter");
        break;
    case half_dollar:printf("half_dollar");
        break;
    case dollar: printf("dollar");
}
```

Diretivas de Pré-processamento

- No C / C++ são necessárias diretivas para guiar o pré-processamento dos arquivos.
- Isso se dá utilizando palavras chaves adequadas:

#define	#elif	#else	#endif
#error	#if	#ifdef	#ifndef
#include	#line	#pragma	#undef

C++

- Observe o seguinte programa em C++:

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

int main()
{
    float f;
    char str[80];
    double d;
    cout << "Enter two floating point numbers: ";
    cin >> f >> d;
    cout << "Enter a string: ";
    cin >> str;
    cout << f << " " << d << " " << str;
    return 0;
}
```

Classes

- Classes definem a natureza de um objeto, suas propriedades e ações que ele consegue executar.

- Para definir a classe usa-se a seguinte sintaxe:

class *class-name {*

private data and functions

access-specifier:

data and functions

access-specifier:

data and functions

// ...

access-specifier:

data and functions

} object-list;

Classes

- Os modificadores de acesso podem ser:

  public

  private

  protected

Classes

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

class employee {
     char name[80]; // private by default
public:
     void putname(char *n); // these are public
     void getname(char *n);
private:
     double wage; // now, private again
public:
     void putwage(double w); // back to public
     double getwage();
};

void employee::putname(char *n) { strcpy(name, n);}
void employee::getname(char *n) {strcpy(n, name);}
void employee::putwage(double w) {wage = w;}
double employee::getwage() {return wage;}
```

Classes

```
int main()
{
    employee ted;
    char name[80];
    ted.putname("Ted Jones");
    ted.putwage(75000);
    ted.getname(name);
    cout << name << " makes $";
    cout << ted.getwage() << " per year.";
    return 0;
}
```

Palavra-chave *friend*

```
#include <iostream>
using namespace std;
class myclass
{
          int a, b;
     public:
          friend int sum(myclass x);
          void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j) { a = i;
 b = j; }
```

```
// Note: sum() is not a member function of any class.
int sum(myclass x)
{
     return x.a + x.b; /* Because sum() is a friend of
                          myclass, it can directly access a and b. */
}
int main()
{
     myclass n;
     n.set_ab(3, 4);
     cout << sum(n);
     return 0;
}
```

Palavra-chave *inline*

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{ return a>b ? a : b; }
int main()
{
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

Substitui na compilação

```
#include <iostream>
using namespace std;
int main()
{
    cout << (10>20 ? 10 : 20);
    cout << " " << (99>88 ? 99 : 88);
    return 0;
}
```

Palavra-chave *inline*

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
// automatic inline
    void init(int i, int j) { a=i; b=j; }
    void show() { cout << a << " " << b << "\n"; }
};
```

```
int main()
{
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
}
```

Construtores

- Construtores são métodos de uma classe que são executados quando um objeto de uma classe é criada.
- Sua assinatura é bem característica:

class_name::class-name()

- Eles não possuem valor de retorno e tem como nome o próprio nome da classe.

Construtores

- Construtores podem ter um ou mais parâmetros.

```
#include <iostream>
#include <cstring>
using namespace std;
const int IN = 1;
const int CHECKED_OUT = 0;
class book {
      char author[40];
      char title[40];
      int status;
public:
      book(char *n, char *t, int s);
      int get_status() {return status;}
      void set_status(int s) {status = s;}
      void show();
};
book::book(char *n, char *t, int s)
{
      strcpy(author, n);
      strcpy(title, t);
      status = s;
}
void book::show()
{
      cout << title << " by " << author;
      cout << " is ";
      if(status==IN) cout << "in.\n";
      else cout << "out.\n";
}
int main()
{
      book b1("Twain", "Tom Sawyer", IN);
      book b2("Melville", "Moby Dick",
CHECKED_OUT);
      b1.show();
      b2.show();
      return 0;
}
```

static

- Pode-se definir variáveis estáticas à uma classe (sendo assim essa variável atrelada a classe e não ao objeto) e/ou à métodos (sendo um método de classe e não de um objeto em específico).

Destrutores

- Assim como um construtor é chamado quando se cria um objeto, um destrutor é chamado quando esse objeto é removido da memória.
- A sintaxe é:

class_name::~class-name()

Objetos em C++

- É possível alocar-se dinamicamente objetos em C++.
- Por isso, compreender como ponteiros funcionam é fundamental.

Ponteiros

- Ponteiros apontam para um local específico da memória.
- Enquanto o *stack* armazena valores e ponteiros, o *heap* armazena os dados voláteis.
- Sendo assim, quando se cria um ponteiro, armazena-se no *stack* o endereço da memória (no *heap*) que guarda o objeto em questão.

Palavra-chave *this*

- Um ponteiro muito importante é o *this*. Ele significa que se está acessando a variável ou método daquele objeto em questão.

```
pwr::pwr(double base, int exp)
{
    this->b = base;
    this->e = exp;
    this->val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--)
        this->val = this->val * this->b;
}
```

Alocação Dinâmica

- Para se alocar dinamicamente uma variável utiliza-se a palavra-chave *new*.
- Da mesma forma, para destruir um objeto dinamicamente utiliza-se a palavra-chave *delete*.

p_var = **new** *type*;

delete *p_var*;

Alocação Dinâmica

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
    int *p;
    try {
        p = new int (87); // initialize to 87
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }
    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";
    delete p;
    return 0;
}
```