Universidade Federal do Ma
Curso de Ciência da Computação

Estrutura de Dados
Prof. Anselmo C. de Paiva
Depto de Informática

Princípios de Programação

Princípios de Programação
1 - Estilo de Programação

- **Conjunto de regras usadas ao codificar um programa com o propósito de facilitar a leitura do código.**
- **Escrever bem leva a um código mais correto.**
- **Base: bom senso e experiência, aliadas a consistência.**
- **Características de um bom estilo de código:**
  - **lógica direta**
  - **expressão natural**
  - **uso convencional da linguagem**
  - **nomes significativos**
  - **formatação simples**
  - **comentários úteis**

Princípios de Programação
Est. de Progr. - Nomes

- **Nome de uma variável ou função rotula um objeto e veicula informações sobre seu propósito.**
- **Deve ser informativo, conciso, memorizável e, se possível, pronunciável.**
- **Quanto mais amplo o escopo mais informações devem ser veiculadas no nome.**
- **Convenções de nomeação facilitam a compreensão do código e a invenção de novos nomes a medida que o código é escrito.**

Princípios de Programação
Est. de Progr. - Nomes

- **Use nomes descritivos para variáveis globais e nomes curtos para variáveis locais**

Globais:
```
int numEntradas;
```

Locais:
```
int n;
int nPontos;
int numeroDePontos; !!exagero
```

- **Variáveis globais usadas de maneira tradicional requerem nome curto.**
  - **i, j contadores de loop**
  - **p,q ponteiros**
  - **s, t strings**

```
for(indiceElemento=0; indiceElemento<numeroElementos; indiceElemento++) {
   vetorElementos[indiceElemento] = indiceElemento;
}
for(i=0;i<numElem;i++) {
   elem[i] = i;
}
```


Princípios de Programação
Est. de Progr. - Nomes

- **Seja consistente: Dê a coisas relacionadas nomes relacionados que mostram seu relacionamento e destaquem suas diferenças.**

```
typedef struct _Fila_ {
   int noDeltensNaF,
      primeiroDaFila,
      filaCapacidade;
} Fila;
```

```
typedef struct _Fila_ {
   int numItens,
      primeiro,
      capacidade;
} Fila;
```

- **Use nomes ativos para função: nomes de função devem se basear em verbos ativos, que podem ser seguidos por substantivos**

```
agora = pegaHora();   numElem = RetornaNumElems();
```

- **Funções que retornam valores lógicos devem ser renomeadas para evitar ambiguidades**

```
if( checkOcta(c)) ....        If( eOcta(c)) .....
```


Princípios de Programação
Est. de Progr. - Nomes

- **Seja preciso: um nome não apenas rotula, ele veicula informações para o leitor. Um nome enganoso pode resultar em *bugs* difíceis de resolver.**

```
#define eoctal(c) ( (c) >='0' && (c) <= '7')

#define eoctal(c) ( (c) <'0' && (c) > '7')
```

Est. de Progr.- Expressões. e Decl.

- **Escreva expressões de maneira clara.**
- **Use espaços nos operadores p/ sugerir agrupamento**
- **Recue para mostrar a estrutura**

```
for(n++;n<100;campo[n++]=\0');   for(n++;n<100;campo[n++]=' \0') {      for(n++;n<100;n++){
*i = \0'; return('\n');                ;                                    campo[n++]=' \0';
                                       }                                    }
                                       *i = \0';                            *i = \0';
                                       return('\n');                        return('\n');
```

- **Use a forma natural das expressões:escreva expressões como vc as falaria em voz alta.**

Expressões condicionais com negações são sempre difíceis de entender

```
if( !(blocoId < blocoAtivo) || !(blocoId >= numBlocos)) ...

if( blocoId >= blocoAtivo) || (blocoId < numBlocos)) ...
```


Est. de Progr.- Expressões e Decl.

- **Use parênteses para resolver as ambiguidades, não confie na ordem de precedência dos operadores**

```
if(x&MASCARA == BITS)   significa:
        if(x&(MASCARA == BITS)) ou  if((x&MASCARA) == BITS))
```

- **Mesmo quando não necessários os parênteses podem ajudar se o agrupamento da expressão for difícil de entender**

```
ano = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
ano = ((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0);
```

- **Divida as expressões complexas**

```
*x +=(*xp=2*k < (n-m)? c[k+1] : d[k--]

if(2*k < (n-m)) {
   *xp = c[k+1];
} else {
   *xp : d[k--];
}
```


Est. de Progr.- Expr. e Decl.

- **Seja Claro**
- **O operador ?: pode levar a construções misteriosas**

```
child = (!LC&&!RC)?0:(!LC?RC:LC);
```

é mais fácil de ler assim:

```
if(LC == 0 && RC == 0 ) {
        child = 0;
}else if (LC == 0 ) {
   child = RC
} else {
   child = LC
}
```

- **É bom para expressões curtas, onde ele pode substituir quatro linhas do *if*, como no código abaixo**

```
max = (a > b) ? a : b;
```

- **Cuidado com os efeitos colaterais**

```
str[i++] = str [i++];
vetor[i++] = i;
scanf("%d %d", &yr, &teste[yr]);
```


Est. de Prog.-Consistência e Idiomas

- **Se o mesmo cálculo é feito da mesma maneira sempre que aparece. Toda variação sugere uma diferença genuína e digna de nota.**
- **Use um estilo de identação e chaves consistente**
  - **Ao longo de nosso curso usaremos o estilo descrito nas regras de submissão de programas.**
- **Use idiomas pra ter consistência: as linguagens de programação tem maneiras convencionais pelas quais os programadores experientes escrevem código comun.**

```
i=0;                          for(i=0; i<n; ){              for( i = 9; i < n; i++) {
while(i < n -1) {                array[i++] = 1.0;             array[i] = 1.0;
   array[i++] =1.0;           }                             }
}
                                                          forma idiomática mais usada
```

- **O Recuo tambem deve ser idiomático**

Est. de Prog.- Macros de função

- **Justificativa: overhead de uma chamada de função.**
- **Hoje em dia isso é irrelevante e suas desvantagens superam em muito seus benefícios.**
- **Evite macros de função.**
- **Problemas:**
  - **parâmetro que aparece mais de uma vez na definição poderia ser avaliado mais de uma vez.**

```
#define isupper(c)  ((c) >= 'A' && (c)  <= 'Z')
chamada assim:  while (isupper(c=getchar())) ...
```

toda vez que um caracter de entrada maior ou igual a A for lido, ele será descartado e outro caractere será lido para ser testado com relação a Z.

- **Se for inevitável usar macros coloque o corpo e os argumentos da macro entre parênteses.**

```
1/square(x)     #define square(x) (x) * (x)      1/ (x) * (x)
                #define square(x) ((x) * (x))    1/((x)*(x))
```


Est. de Prog.- Números mágicos

- **São as constantes, os tamanhos de vetores, as posições de caracteres, fatores de conversão e outros valores numéricos literais que aparecem nos programas.**
- **Dê nome aos números mágicos: todo numero diferente de 0 ou 1 pode ser mágico e deve receber seu próprio nome.**
- **Exemplo: Trecho de programa para calcularum histograma de frequência de letras em terminal com 24x80**

```
fac = lim/20;
if(fac < 1)   fac = 1;
/* gerar histograma */
for(  i = 0, col = 0; i < 27; i++, j++) {
  col += 3;
  k=21-(let[i]==0);
  star = (let[i] == 0 ) ? ' ' : '*';
  for(j=k; j<22; j++)
    draw(j, col, star);
}
```

```
draw(23, 2, ' ');  /* eixo x de rótulo */
for(i='A'; i<= 'Z';i++)
  printf("%c ", i);
```

Est. de Prog.- Números Mágicos

O código inclui entre outros números 20, 21, 22, 23 e 27

Podemos verificar que existem três números críticos

24 - número de linhas da tela; 80 - número de colunas e

26 - número de letras do alfabeto

O código então fica assim:

```
enum {
  MINROW = 1,          /*lado superior */
  MINCOL = 1,          /* lado esquerdo */
  MAXROW = 24,         /*lado inferior (<=) */
  MAXCOL = 80,         /*lado direito (<=) */
  LABELROW = 1,        /*posição dos rótulos */
  NLET = 26,           /*tamanho do alfabeto */
  HEIGHT = MAXROW - 4, /* altura das barras */
  WIDTH = (MAXCOL-1)/NLET /* larg. barras */
}
fac = (lim+ HEIGHT -1)/HEIGHT;
if(fac < 1) {
  fac = 1;
}
```

```
for( i = 0; i < NLET; i++) {  /* gerar histograma */
  if ( let[i] != 0 ) {
    for (j=HEIGHT -let[i]*fac; j<HEIGHT; j++) {
      draw(j+1 + LABELROW, (i+1)*WIDTH, '*');
    } /*rof */
  } /* fi */
} /* rof */

draw(MAXROW -1, MINCOL+1,' '); /* eixo x de rótulo */

for(i='A'; i<= 'Z';i++) {
  printf("%c ", i);
}
```


Est. de Prog.- Números Mágicos

- **Defina os números mágicos inteiros como *enum*.Use o *#define* apenas para constantes reais**
- **Use constantes de caracteres não inteiros**

```
if ( c >= 65 && c <= 90 ) ...

if ( c >= 'A' && c <= 'Z' ) ...
```

- **Use a linguagem para calcular o tamanho de um objeto**

  Não use nenhum tamanho explícito para tipos de dados

  Use sempre *sizeof(int)* ao invés de 2 ou 4

  Para calcularn o tamanho de um vetor em C use a seguinte macro:

```
#define NELEMS(vetor) (sizeof(vetor)/sizeof(vetor[0]))
```

  este é um uso apropriado para uma macro porque faz algo que uma função não faria, calcular o tamanho de um vetor a partir de sua declaração.

Est. de Prog.- Comentários

- **Destinam-se a ajudar o leitor de um programa.**
- **Não enfatize o óbvio:**

```
/* default */      /* retorna SUCCESS */     /* incrementa o contador de entrada zero */
default:           return SUCCES;             zerocount++
  ...
```

- **Comente funções, dados globais, definições de constantes, campos de estruturas, e tudo que o comentário possa auxiliar na compreensão.**

```
typedef struct _state_ {     /*prefixo + lista de sufixos */
  char   pref[NPREF];        /*palavras de prefixo */
  Suffix *suf;               /* lista de sufixos */
  State  *next;              /* próximo da tabela */
} State;
```

- **Um comentário que apresenta cada função, prepara a leitura do próprio código.**

```
/* random : retorna um inteiro na faixa [0..r-1] */
int random( int r ) {
  ...
```


Est. de Prog.- Comentários

- **Código realmente difícil, em face de um algoritmo complicado ou a estruturas de dados intricadas, devem conter um comentário que indique uma referência que possa auxiliar o leitor**

```
/*
 * idct: Implementação da transformada discreta do cosseno.
 * Algoritmo de Chen-wang ( IEEE ASSP-32, págs. 803-816, Agosto de 1984).
 */
static void idct (int b[8*8])
{
  ...
```

- **Não comente código ruim rescreva-o.**
- **Não contradiga o código**

  Ao solucionar um bug não se esqueça de verificar se os comentários estão de acordo com as suas modificações no código

2 - Depuração

- *Bug* **"Um defeito ou falha em uma máquina, plano, ou similar. Origem EUA 1889 Paull Mall Gaz. ...**

  (Oxford English Dictionary, 2nd Edition)

- **Passa-se tanto tempo depurando qto escrevendo bons programas.**
- **Técnicas para reduzir tempo de depuração:**
  - **bom projeto;**
  - **bom estilo;**
  - **testes de condições limites;**
  - **programação defensiva;**
  - **ferramentas de verificação;**

Depuração - Boas pistas, bugs fáceis

- **Procure padrões conhecidos**

```
int n;                  int n;
scanf("%d",n);          scanf("%d", &n);
```

- ***Examine a alteração mais recente***
  - ***preserve a versão correta anterior do código;***
  - ***mantenha registro das alterações feitas;***
  - ***procure usar um sistema de controle de código(Ex. RCS, SourceSafe, etc.)***
- ***Não cometa o mesmo erro duas vezes***
  - ***verifique se a mesma construção errada não foi usada em outros locais do código***
- ***Depure agora, não mais tarde***
  - ***mesmo que o bug não lhe atrapalhe, procure-o logo***

Depuração - Boas pistas, bugs fáceis

- **Obtenha um rastreamento de pilha**
  - **depuradores possibiltam exame do estado pós-orte do programa;**
  - **número da linha de código onde ocorreu a FALHA (ERRO)**
- **Leia antes de digitar**
  - **leia e analize o código fonte antes de tentar fazer alterações para encontrar o problema;**
- **Explique seu código para outra pessoa**

Depuração - Boas pistas, bugs fáceis

- **Procure padrões conhecidos**

```
int n;                  int n;
scanf("%d",n);          scanf("%d", &n);
```

- ***Examine a alteração mais recente***
  - ***preserve a versão correta anterior do código;***
  - ***mantenha registro das alterações feitas;***
  - ***procure usar um sistema de controle de código(Ex. RCS, SourceSafe, etc.)***
- ***Não cometa o mesmo erro duas vezes***
  - ***verifique se a mesma construção errada não foi usada em outros locais do código***
- ***Depure agora, não mais tarde***
  - ***mesmo que o bug não lhe atrapalhe, procure-o***

Depuração - Boas pistas, bugs fáceis

- **Obtenha um rastreamento de pilha**
  - **depuradores possibiltam exame do estado pós-orte do programa;**
  - **número da linha de código onde ocorreu a FALHA (ERRO)**
- **Leia antes de digitar**
  - **leia e analize o código fonte antes de tentar fazer alterações para encontrar o problema;**
- **Explique seu código para outra pessoa**

Depuração -Sem pistas, bugs difíceis

- **Torne o bug reproduzível**
  - **gaste algum tempo construindo entrada e definições de parâmetro que certamente causarão o problema;**
- **Divida e conquiste**
  - **cri o menor exemplo possível que reproduz o bug**
- **Utilize o depurador** *printf*
  - **coloque impressões de mensagens "passei por aqui", ou com a impressão de resultados intermediários no meio do código;**
- **Escreva código autoverificador**
  - **caso seja necessário escreva funções que testam se determinado estado do programa esta ok.**
- **Desenhe uma figura**
- **Use as ferramentas**
- **mantenha registros das alterações**

Depuração -Bugs não reproduzidos

- **Verifique se todas as variáveis foram inicializadas**
- **Se o erro muda de comportamento ou desaparece qdo código de depuração for adicionado, pode ser um erro de alocação de memória**
- **Quando um programa funciona com uma pessoa e falha pra outra: pode ser o ambiente externo:**
  - **arquivos lidos**
  - **permissões de acesso aos arquivos**
  - **caminho de pesquisa dos comandos(path)**

3 - Testes

- **Depuração: aquilo que vc faz quando sabe que um programa está com problemas.**
- **Teste: uma tentaiva determinada e sistemática de quebrar um programa que você acha que esta funcionando.**
- **"Testes podem demonstrar a presença de um bug mas não sua ausência" (Edsger Dijkstra)**

Testes -Teste enqto codifica

- Qdo mais cedo um problema for encontrado melhor ;
- Teste o código nos seus limites: "Teste da condição limite"
  - a medida que cada parte do código é escrita verifique logo se a condição se ramifica na direção certa ou se o loop tem o número adequad de repetições.
- Teste as pré e pós condições
  - verifique se as propriedades esperadas ou necessárias mantidas antes(precondição) ou depois (pós-condição) de alguma parte do código são executadas.
  - Verifique se os valores de entrada são aceitáveis
- Use afirmações (*assert*)

  assert(n>0) Assertion failed: n>0, file avgtest.c, line 7
- Verifique o retorno de erro

Testes - Teste sistemático

- Teste incrementalmente
- Teste primeiro as partes mais simples

Ex: Teste sistematico de uma f unção que executa pesquisa binária em um vetor de inteiros
  - vetor sem elementos
  - vetor com um elemento e um valor de pesquisa:
    - menor que o elemento único do vetor
    - igual ao elemento único do vetor
    - maior ao elemento único do vetor
  - vetor com dois, três e quatro elementos e valores experimentais que testem as várias posições possíveis (cinco para o vetro com dois elementos)
  - vetor com elementos repetidos e valor de pesquisa:
    - menores, iguais ou maiores que os valores do vetor

4 - Desempenho

- Quando devemos agilizar um programa?
  - Se preocupe com o desempenho somente se o problema for importante e houve possibilidade dele ficar mais rápido mantendo a mesma robustez, correção e clareza.
  - Primeiro princípio: "Não otimize"
- Como podemos fazer isso?
- Quais ganhos podemos esperar?

Desempenho - Sincronização e perfil

- Automatize as medições de sincronização

```
% time slowprogram      clock_t before;
real 7.0                double elapsed;
user 6.2                before = clock();
sys  0.1                slow_function();
                        elapsed = clock() - before;
                        printf("function used %.3f seconds\n",
                               elapsed/CLOCKS_PER_SEC);
```

- Use um profiler - gerador de perfis de programas
  - perfil - medida de onde o programa gasta seu tempo
  - util para determinar os gargalos do programa
  - "Menos de 4% de um programa geralmente representam mais da metade de seu tempo de execução" (Don Knuth)

```
% cc -p spamtestc -o spamtest
%spamtest
prof spamtes
```


Desempenho - Sincronização e perfil

- Concentre-se nos gargalos
- Desenhe uma figura

Desempenho – Estrat. p/ Velocidade

- Use um algoritmo ou estrutura de dados melhores
- Ative as opções de otimização do compilador
- Ajuste o código
  - Colete as subexpressões comuns `sqrt(dx*dx + dy* dy)+ ((sqrt(dx*dx + dy*dy))....`
  - Susbtitua as operações caras pelas baratas
    - divisão e resto muit mais lento que multiplicação
    - Ex. Distância entre dois pontos.
  - Desenrole ou elimine os loops: se o corpo do loop nao for longo demais e não interagir muitas vezes..
  - Faça o cache dos valores mais usados
  - Pré-calcule os resultados
  - Reescreva em uma linguagem de nível inferior
- Não otimize o que não tem importância

Princípios de Programação
4.3 - Estimativa

- **Exemplo de resultados (MIPS R10000 de 250 MHZ)**

Operações int		Operações double		Estruturas de Controle	
i1++	8	d1 = d2	8	if(i==5) i1++;	4
i1 = i2 + i3	12	d1 = d2 + d3	12	if(i!=5) i1++;	12
i1 = i2 - i3	12	d1 = d2 - d3	12	while(i<0) i1++	3
i1 = i2 * i3	12	d1 = d2 * d3	11	i1 = sum1(12)	57
i1 = i2 / i3	114	d1 = d2 / d3	58	i1= sum2(i2,i3)	58
i1 = i2 % i3	114	Conversões numéricas		i1 = sum3(i2,i3,i4)	54
Operações float		i1 = f1	8		
f1 = f2	8	f1 = i1	8		
f1 = f2 +fi3	12	Operações de vetor inteiro			
f1 = f2 - f3	12	v[i] = i	49		
f1 = f2 * f3	11	v[v[i]] = i	81		
f1 = f2 / f3	28	v[v[v[i]]] = i	100		

Princípios de Programação
Desempenho -Estimativas

Exemplo de resultados (MIPS R10000 de 250 MHZ)

Entrada/Saída		Conversões string/número	
fputs(s, fp)	270	i1 = atoi("12345")	402
fgets(s,9,fp)	222	sscanf("12345", "%d", &i1)	2376
fprintf(fp,"%d\ n",i)	1820	sprintf(s,"%d", i)	1492
fscanf(fp,"%d", &I)	2070	f1 = atof("123.45")	4098
		sscanf("123.45", "%f", &f1)	6438
Malloc		sprintf(s, "6.2f", 123.45)	3902
free(malloc(8))	342	Funções matemáticas	
		i1= rand()	135
Funçõesde string		f1 = log(f2)	418
strcpy(s,"0123456789")	157	f1 = exp(f2)	462
i1 = strcmp(s,s)	176	f1 = sin(f2)	514
i1 = strcmp(s,"a123456789")	64	f1 = sqrt(f2)	112

Princípios de Programação
Testes - Teste sistemático

- **Saiba qual saída deve esperar**
- **Verifique as propriedades de conservação**

Princípios de Programação
Leitura Suplementar

Esta seção é completamente baseada no livro:

Kernighan, B. W. e Pike, R. "A Prática da Programação", Ed. Campus, Rio de Janeiro, 1999.

Outras Referências sobre o Assunto:

Kernighan, B. W. e Plauger, P. J. "The Elements of Programming Style", McGraw-Hill, 1993.

Maguire, S. "Writing Solid Code"Microsoft Press, 1993.