Alguns conceitos de C++ que usaremos neste curso (para quem já conhece C)

Referência "&".

Suponha que se queira criar uma função que soma 1 numa variável inteira.

1) A seguinte solução está errada:

```
#include <cstdio>

void somaum(int k)
{ k=k+1; }

int main()
{ int z=4;
  somaum(z);
  printf("z=%d",z);
}
```

A variável *z* continuará valendo 4 após a chamada da função *somaum*. A passagem de parâmetro *k* foi "por valor". Isto é, foi copiada o valor de *z* para a variável *k*. No final da função somaum, a variável *k* é destruída e a variável *z* continua com o valor inalterado.

2) A solução em C para fazer a função *somaum* funcionar corretamente é:

```
#include <cstdio>

void somaum(int* k)
{ *k=*k+1; }

int main()
{ int z=4;
  somaum(&z);
}
```

Aqui, está passando o endereço de *z* (&*z*) para somaum, cujo argumento é um apontador para uma variável inteira *k*. O programa funciona, mas não fica "limpo". Quanto mais confuso um programa, maior é a possibilidade de se cometer um erro de programação.

3) A solução em C++ para fazer a função *somaum* funcionar corretamente é:

```
#include <cstdio>

void somaum(int& k)
{ k=k+1; }

int main()
{ int z=4;
  somaum(z);
  printf("z=%d",z);
}
```

Este programa faz exatamente a mesma coisa que a solução 1.2 e ficou muito mais "limpo". Basta colocar o operador de referência & para que tudo funcione.

Mecanismos de cópia de imagens

OpenCV/Cekeikon possui mecanismo "estranho" de cópia:

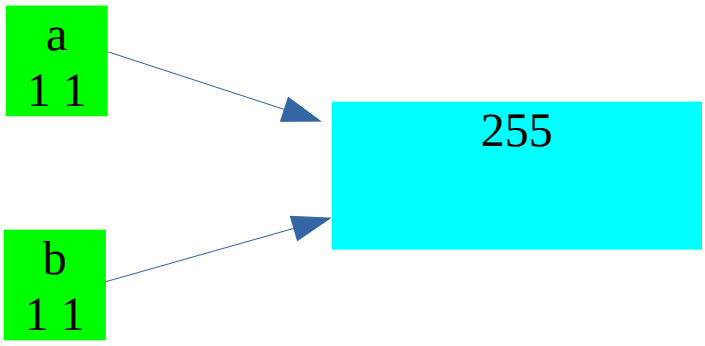
1) Em OpenCV/Cekeikon, “copy constructor” e “copy operator” copiam somente os cabeçalhos que apontam para uma mesma imagem:

```
#include <cekeikon.h>
int main()
{ Mat_<GRY> a(1,1, 255);
  Mat_<GRY> b;
  b=a;
  a(0,0)=0;
  cout << a;
  cout << b;
}
```

saida:

```
1 1 // rows, cols de a
  0 // valor de a

1 1 // rows, cols de b
  0 // valor de b
```

Note que o valor de b alterou-se, mesmo sem ter feito alteração explícita.



3) Em OpenCV/Cekeikon, quando necessário, o programador deve clonar uma imagem para forçar a fazer cópia de uma imagem.

```
#include <cekeikon.h>
int main()
{ Mat_<GRY> a(1,1, 255);
  Mat_<GRY> b;
  b=a.clone(); // ou a.copyTo(b);
  a(0,0)=0;
  cout << a;
  cout << b;
}
```

saida:

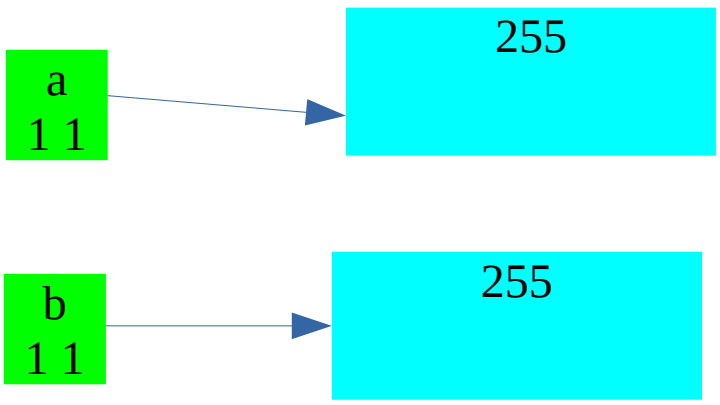
```
1 1 // rows, cols de a
  0 // valor de a

1 1 // rows, cols de b
255 // valor de b
```

Nota: Em C++, 0 (zero) tem vários significados. Assim, a construção:

```
Mat_<GRY> a(1,1, 0);
```

dá erro, pois é ambíguo. Para resolver a ambiguidade, tem que escrever:

```
Mat_<GRY> a(1,1, GRY(0));
```



Passagem de parâmetro (imagens com/sem &)

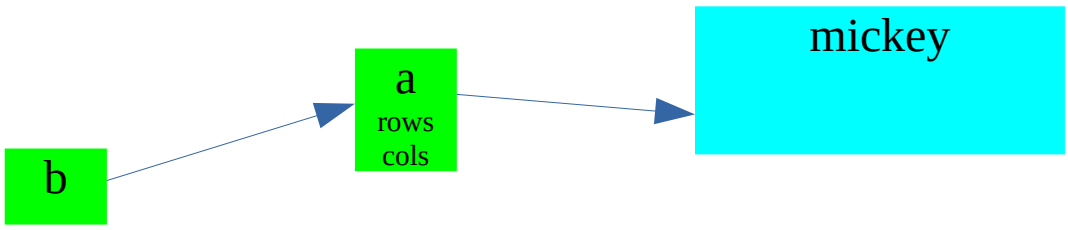
1) Uma função que cria a imagem negativa em OpenCV/Cekeikon.

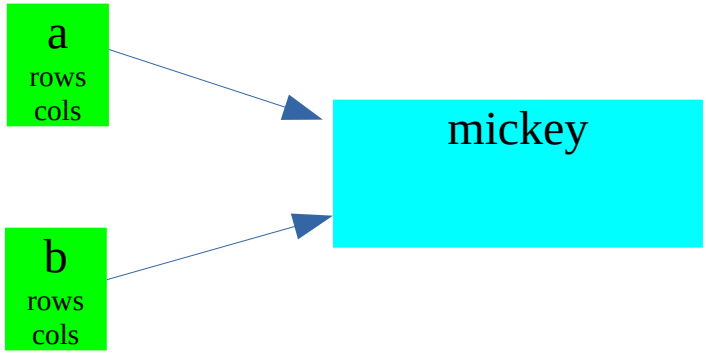
```
#include <cekeikon.h>

void negativa(Mat_<GRY>& b)
{ for (int l=0; l<b.rows; l++)
    for (int c=0; c<b.cols; c++)
      if (b(l,c)==0) b(l,c)=255;
      else b(l,c)=0;
}

int main()
{ Mat_<GRY> a;
  le(a,"mickey_reduz.bmp");
  negativa(a);
  mostra(a);
}
```

Esta função funciona com ou sem o operador referência &.

Nota: Se uma imagem “Mat_<???> a” for passada sem o operador de referência, então o seu tamanho não poderá ser alterada dentro da função. Porém, os conteúdos dos pixels da imagem podem ser alteradas.

Nota: Se uma imagem “Mat_<???>& a” for passada com o operador de referência, então o seu tamanho (assim como os conteúdos dos pixels) poderá ser alterada dentro da função.





Header, módulos.

Em C ou C++, é possível dividir um programa grande em pequenos módulos. Considere o programa somaum.cpp:

```
#include <cstdio>

void somaum(int& k)
{ k=k+1; }

int main()
{ int z=4;
  somaum(z);
}
```

Este programa poderia ser dividido em dois módulos: função somaum e o programa principal:

```
//somaum.cpp
#include <cstdio>

void somaum(int& k)
{ k=k+1; }
```

```
//principal.cpp
#include <cstdio>

void somaum(int& k);

int main()
{ int z=4;
  somaum(z);
}
```

A sequência de comandos para compilar os dois módulos e linká-los é:

```
g++ -c somaum.cpp -o somaum.o
g++ -c principal.cpp -o principal.o
g++ -o principal.exe principal.o somaum.o
```

Melhor ainda, ser subdividido em dois módulos com o cabeçalho da função somaum:

```
//somaum.cpp - definição da função
#include <cstdio>

void somaum(int& k)
{ k=k+1; }
```

```
//somaum.h - declaração da função
void somaum(int& k);
```

```
//principal.cpp
#include <cstdio>
#include "somaum.h" // inclui declaração no diretorio local

int main()
{ int z=4;
  somaum(z);
}
```

Sequência de comandos para compilar os dois módulos e linká-los:

```
g++ -c somaum.cpp -o somaum.o
g++ -c principal.cpp -o principal.o
g++ -o principal.exe principal.o somaum.o
```

Neste caso, os módulos somaum e principal podem ser compilados separadamente. Se alterar alguma coisa, somente o módulo alterado precisa ser recompilado. Link junta os módulos compilados separadamente.

O programa “make” pode ser usada para verificar automaticamente quais foram os módulos alterados (pelo horário de alteração dos arquivos .cpp e .o) e recompilar somente os arquivos necessários.

Para criar a biblioteca Proeikon, separei as funções em módulos e compilei-os. O conjunto de módulos compilados (com cabeçalhos respectivos) é a biblioteca.

Se quiserem ver a lista de todas as funções da biblioteca, podem olhar os arquivos header .h.

3) Leitura dos argumentos

Queremos fazer um programa que recebe dois números e imprime a soma deles.

```
c:\lixo>soma 4 7
4+7=11
c:\lixo>
```

Há uma forma padrão de fazer isto em C/C++:

```
//soma.cpp
#include <cstdio>

int main(int argc, char** argv)
{ if (argc!=3) {
    printf("soma n1 n2\n");
    exit(0);
  }
  int n1; sscanf(argv[1],"%d",&n1);
  int n2; sscanf(argv[2],"%d",&n2);
  printf("%d+%d=%d\n",n1,n2,n1+n2);
}
```

4) Leitura de argumentos com expansão de “wildcard”

A maioria dos compiladores faz “expansão de wildcard” dos argumentos do programa. Por exemplo, o programa argument.cpp abaixo imprime todos os argumentos do programa.

```
#include <cekeikon.h>
int main(int argc, char** argv)
{ for (int i=0; i<argc; i++)
    printf("Argumento %d=%s\n",i,argv[i]);
}
```

Se compilar esse programa e executar:

```
c:\>argument *.jpg
```

O programa irá listar todos os arquivos do tipo JPG do diretório local.

Você pode desligar a expansão de wildcard colocando o seguinte comando no início do programa:

```
#include <cekeikon.h>
int _CRT_glob = 0;   // Para impedir expansao de wildcard
... resto do programa
```

A função vsWildCard do Cekeikon pode ser usada para fazer a expansão:

```
    void vsWildCard(string nome, vector<string>& vs)
```

Se chamar:

```
    vector<string> vs;
    vsWildCard(“*.jpg”, vs);
```

O vetor de string vs irá conter, em cada entrada, o nome de um arquivo tipo JPG do diretório local.

"Debugar" programa:

Prefiro não usar debugador. Coloco “printf” ou “mostra” ou “cout <<” ou ... no programa para debugar o programa.

Para ajudar no processo de debug, criei os macros

```
xdebug
xdebug1("Mensagem")
```

Esses macros imprimen o nome do arquivo e a linha dentro do arquivo.

Exemplo:

```
1   #include <cekeikon.h>
2   int main() {
3     xdebug1("Inicio do programa");
4     Mat_<GRY> a(1,1, 255);
5     Mat_<GRY> b;
6     xdebug;
7     b=a.clone(); // ou a.copyTo(b);
8     xdebug;
9     int zero=0;
10    a(0,0)=a(1,1)/zero;  //erro aqui
11    xdebug;
12    cout << a;
13    cout << b;
14    xdebug1("Cheguei no final do programa");
15  }
```

O programa acima vai abortar na linha 10. Ele imprime:

```
>debuga
File=debuga.cpp line=3 Inicio do programa
File=debuga.cpp line=6
File=debuga.cpp line=8
Floating point exception
```

indicando que executou corretamente até a linha 8 e não chegou na linha 11.

Redirecionamento de entrada/saída:

Tanto no Windows como no Linux, há a possibilidade de redirecionar entrada/saída. Considere o seguinte programa:

```
//somaum.c
#include <stdio.h>
int main() {
   int i;
   scanf("%d",&i);
   printf("%d\n",i+1);
}
```

Este programa lê um número inteiro, soma um, e imprime o resultado. Rodando no terminal, o programa espera que o usuário tecle um número, e imprime o número mais um.

```
$somaum
2  <= o número que o usuário digitou
3  <= o número que o computador imprimiu
```

É possível ler a entrada do arquivo em vez do teclado (o arquivo texto "dois.txt" contém o número dois):

```
$somaum <dois.txt
3  <= o número que o computador imprimiu
```

É possível redirecionar a saída para um arquivo:

```
$somaum <dois.txt >tres.txt
```

Aqui, o arquivo tres.txt terá o número três. A saída não irá para terminal.

Usando "pipe", a saída de um programa torna-se a entrada do outro.

```
$somaum | somaum
2 <= o número que o usuário digitou
4 <= o número que o computador imprimiu

$somaum <dois.txt | somaum | somaum
5 <= o número que o computador imprimiu


Overwrite
    > - standard output
    < - standard input
    2> - standard error

Append
    >> - standard output
    << - standard input
    2>> - standard error
```