HALCON Version 6.1.4

MVTec Software GmbH

# HALCON / C++

User's Manual

How to use the image analysis tool HALCON, Version 6.1.4, in your own C++ programs

| | | |
|---|---|---|
| Edition 1 | July 1997 | |
| Edition 2 | November 1997 | |
| Edition 3 | March 1998 | (HALCON 5.1) |
| Edition 4 | April 1999 | (HALCON 5.2) |
| Edition 5 | October 2000 | (HALCON 6.0) |
| Edition 5a | July 2001 | (HALCON 6.0.1) |
| Edition 5b | February 2002 | (HALCON 6.0.2) |
| Edition 6 | June 2002 | (HALCON 6.1) |
| Edition 6a | December 2002 | (HALCON 6.1.1) |
| Edition 6b | February 2005 | (HALCON 6.1.4) |

More information about HALCON can be found at:

`http://www.mvtec.com/halcon/`

# About This Manual

This manual describes the interface of HALCON to the programming language C++. It provides all necessary information to understand and use the provided C++ classes in your own programs. A set of example programs shows how to apply HALCON/C++ to solve typical image processing tasks.

The reader of this user manual should be familar with basic concepts of image analysis and the programming language C++.

The manual is divided into the following chapters:

- **Introducing HALCON/C++**
  A first example shows how easy image processing becomes using HALCON/C++.
- **Basics of the HALCON/C++ Interface**
  This chapter describes the basics of the HALCON/C++, e.g., how to call HALCON operators in the procedural and object-oriented approach.
- **The HALCON Parameter Classes**
  This chapter lists the parameter classes of HALCON.
- **Creating Applications Using HALCON/C++**
  This chapter explains how to compile and link C++ programs with HALCON/C++.
- **Typical Image Processing Problems**
  This chapter contains example programs for typical image processing tasks.

# Release Notes

Please note the latest updates of this manual:

- **Edition 6b, HALCON 6.1.4 (February 2005)**
  Errors in the example for the class HRegion and in the member function lists of HRegionArray and Hobject have been corrected.

- **Edition 6a, HALCON 6.1.1 (December 2002)**
  An error in the description of object-oriented error handling was corrected. Besides, the manual now warns that the user must allocate memory for output string parameters him/herself. Finally, the manual reflects the support of gcc-3.2.

- **Edition 6, HALCON 6.1 (June 2002)**
  The manual has been revised and extended significantly: The basics of the HALCON/C++ interface, e.g., how to call operators in the procedural and in the object-oriented way or how to realize error handling, are now described in detail in a new chapter; The section “Exception Handling” has been extended by an example showing how to use the C++ exception handling mechanism (`try...catch`) together with the class `HException`. Furthermore, the chapter describing the parameter classes has been updated and now also lists the new handle classes.

- **Edition 5b, HALCON 6.0.2 (February 2002)**
  Errors in the description of the class HException were corrected.

- **Edition 5a, HALCON 6.0.1 (July 2001)**
  Since HALCON 6.0.1 does not support HP-UX anymore, the corresponding references have been deleted from the manual.

- **Edition 5, HALCON 6.0 (October 2000)**
  The manual has been restructured and revised slightly, especially the chapter “Creating Applications Using HALCON/C++”.

# Contents

# Chapter 1

# Introducing HALCON/C++

HALCON/C++ is the interface of the image analysis system HALCON to the programming language C++. Together with the HALCON library, it allows to use the image processing power of HALCON inside C++ programs. HALCON provides operators covering a wide range of applications, e.g., factory automation, quality control, remote sensing, aerial image interpretation, medical image analysis, and surveillance tasks.

After pointing out additional sources of information about HALCON, we start with a first example application. The following chapters describe the use of the HALCON operators in C++ programs in more detail: Chapter 2 takes a closer look at the basics of the HALCON/C++ interface, while chapter 3 gives an overview of the HALCON's parameter classes `HImage`, `HRegion`, `HWindow` etc. Chapter 4 shows how to create applications based on HALCON/C++. Finally, chapter 5 presents typical image processing problems and shows how to solve them using HALCON/C++.

## 1.1 Additional Sources of Information

For further information you may consult the following manuals:

- Getting Started with HALCON
  An introduction to HALCON in general, including how to install and configure HALCON.
- HDevelop User's Manual
  An introduction to the graphical development environment of the HALCON system.
- HALCON/C User's Manual
  How to use the HALCON library in your C programs.
- HALCON/COM User's Manual
  How to use the HALCON library in your COM programs.
- Extension Package Programmer's Manual
  How to extend the HALCON system with your own operators.
- Frame Grabber Integration Programmer's Manual
  A guide on how to integrate a new frame grabber in the HALCON system. Note that in some cases you might define new operators (using the Extension Package Interface)

instead of using the standard HALCON Frame Grabber Integration Interface in order to exploit specific hardware features of a frame grabber board.

- HALCON/C++ , HALCON/HDevelop , HALCON/C , HALCON/COM
  The reference manuals for all HALCON operators (versions for C++, HDevelop, C, and COM).
- **Application Guide**
  Multiple, independent documents called Application Notes, written from the point of view of developing machine vision applications with HALCON. Each Note covers a specific topic, e.g., how to use shape-based matching to find and localize objects.

All these manuals are available as PDF documents. The reference manuals are available as HTML documents as well. For the latest version of the manuals please check

```
http://www.mvtec.com/halcon/
```

## 1.2 A First Example

Let's start with a brief sample program before taking a closer look inside HALCON/C++.

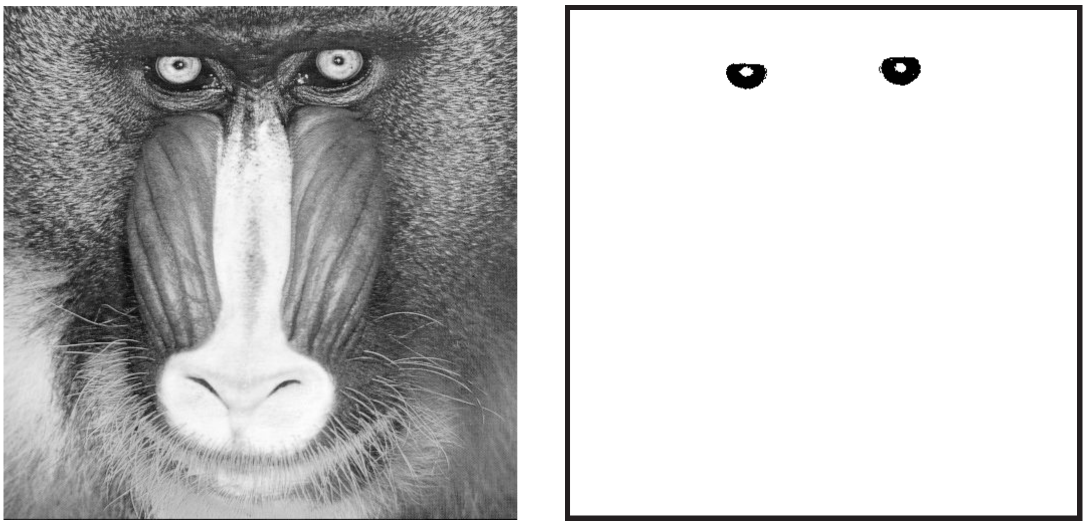

Figure 1.1: The left side shows the input image (a mandrill), and the right side shows the result of the image processing: the eyes of the monkey.

The input image is shown in figure 1.1 on the left side. The task is to find the eyes of the monkey by segmentation. The segmentation of the eyes is performed by the C++ program listed in figure 1.2, the result of the segmentation process is shown in figure 1.1 on the right side.

The program is more or less self-explaining. The basic idea is as follows: First, all pixels of the input image are selected which have a gray value of at least 128, on the assumption that the image `Mandrill` is a byte image with a gray value range between 0 and 255. Secondly, the connected component analysis is performed. The result of the HALCON operator is an array of

```
#include "HalconCpp.h"

main()
{
  HImage  Mandrill("monkey");          // read image from file "monkey"
  HWindow w;                           // window with size equal to image

  Mandrill.Display(w);                 // display image in window
  w.Click();                           // wait for mouse click

  HRegion Bright = Mandrill >= 128;    // select all bright pixels
  HRegionArray Conn = Bright.Connection(); // get connected components

  // select regions with a size of at least 500 pixels
  HRegionArray Large = Conn.SelectShape("area","and",500,90000);

  // select the eyes out of the instance variable Large by using
  // the anisometry as region feature:
  HRegionArray Eyes = Large.SelectShape("anisometry","and",1,1.7);

  Eyes.Display(w);                     // display result image in window
  w.Click();                           // wait for mouse click
}
```

Figure 1.2: This program extract the eyes of the monkey.

regions. Each region is isolated in the sense that it does not touch another region according to the neighbourhood relationship. Among these regions those two are selected which correspond to the eyes of the monkey. This is done by using shape properties of the regions, the size and the anisometry.

This example shows how easy it is to integrate HALCON operators in any C++ program. Their use is very intuitive: You don't have to care about the underlying data structures and algorithms, you can ignore specific hardware requirements, if you consider e.g. input and output operators. HALCON handles the memory management efficiently and hides details from you, and provides an easy to use runtime system.

# Chapter 2

# Basics of the HALCON/C++ Interface

In fact, the HALCON/C++ interface provides two different approaches to use HALCON's functionality within your C++ program: a *procedural* and an *object-oriented* approach. The procedural approach corresponds to calling HALCON operators directly as in C or HDevelop, e.g.:

```
Hobject     original_image, smoothed_image;
::read_image(&original_image, "monkey");
::mean_image(original_image, &smoothed_image, 11, 11);
```

Note that, in comparison with HALCON/C or HDevelop, the operator names are prefixed with a double colon. In the following, we will leave out this prefix when referencing operators in the text.

In addition to the procedural approach, HALCON/C++ allows to call HALCON operators in an object-oriented way, i.e., via a set of classes. For example, the code from above can be "translated" into:

```
HImage      original_image("monkey"), smoothed_image;
smoothed_image = original_image.MeanImage(11, 11);
```

This simple example already shows that the two approaches result in clearly different code: Besides the different operator names (procedural: small letters and underscores; object-oriented: capitals), the operator calls differ in the number and type of parameters. Furthermore, functionality may be available in different ways; for example, images can be read from files via a constructor of the class `HImage`. In general, we recommend to use the object-oriented approach. Note, however, that HDevelop can export programs only into procedural C++ code. Section 2.4 shows how to combine procedural with object-oriented code.

In the following sections, we take a closer look at various issues regarding the use of the HALCON/C++ interface; chapter 3 describes the provided classes in more detail.

## 2.1 Calling HALCON Operators

How a HALCON operator can be called via the HALCON/C++ interface is described in detail in the HALCON/C++ reference manual. As an example, figure 2.1 shows parts of the entry for the operator `mean_image`.

Below, we take a closer look at the parameters of an operator call, describe how to call operators via classes, and explain another special HALCON concept, the *tuple mode*.

```
Herror ::mean_image ( Hobject Image, Hobject *ImageMean,
const HTuple &MaskWidth, const HTuple &MaskHeight )
```

```
HImage HImage::MeanImage ( const HTuple &MaskWidth,
const HTuple &MaskHeight ) const
```

```
HImageArray HImageArray::MeanImage ( const HTuple &MaskWidth,
const HTuple &MaskHeight ) const
```

| | | |
|---|---|---|
| `Image` (input_object) | (multichannel-)image(-array) | ⇝ *Hobject: HImage(Array)* ( byte / int2 / uint2 / int4 / real / dvf ) |
| `ImageMean` (output_object) | (multichannel-)image(-array) | ⇝ *Hobject * : HImage(Array)* ( byte / int2 / uint2 / int4 / real / dvf ) |
| `MaskWidth` (input_control) | extent.x | ⇝ *HTuple.long* |
| `MaskHeight` (input_control) | extent.y | ⇝ *HTuple.long* |

Figure 2.1: The head and parts of the parameter section of the reference manual entry for `mean_image`.

### 2.1.1 A Closer Look at Parameters

HALCON distinguishes two types of parameters: *iconic* and *control* parameters. *Iconic parameters* are related to the original image (images, regions, XLD objects), whereas *control parameters* are all kinds of alphanumerical values, such as integers, floating-point numbers, or strings.

A special form of control parameters are the so-called *handles*. A well-known representative of this type is the *window handle*, which provides access to an opened HALCON window, e.g., to display an image in it. Besides, handles are used when operators share complex data, e.g., the operators for shape-based matching which create and then use the model data, or for accessing input/output devices, e.g., frame grabbers. Classes encapsulating handles are described in detail in section 3.2.3.

Both iconic and control parameters can appear as input and output parameters of a HALCON operator. For example, the operator `mean_image` expects one iconic input parameter, one iconic output parameter, and two input control parameters (see figure 2.1); figure 2.2 shows an operator which has all four parameter types. Note how some parameters "disappear" from within the brackets if you call an operator via a class; this mechanism is described in more detail in section 2.1.2.

An important concept of HALCON's philosophy regarding parameters is that **input parameters are not modified by an operator**. As a consequence, they are passed *by value* (e.g., `Hobject Image` in figure 2.1) or via a constant reference (e.g., `const HTuple &MaskWidth`). This philosophy also holds if an operator is called via a class, with the calling instance acting as an input parameter. Thus, in the following example code the original image is not modified by the call to `MeanImage`; the operator's result, i.e., the smoothed image, is provided via the return value instead:

```
Herror ::find_1d_bar_code ( Hobject Image, Hobject *CodeRegion,
const HTuple &BarCodeDescr, const HTuple &GenericName,
const HTuple &GenericValue, HTuple *BarcodeFound,
HTuple *BarCodeElements, HTuple *Orientation )
```

```
HRegion HImage::Find1dBarCode ( const HTuple &BarCodeDescr,
const HTuple &GenericName, const HTuple &GenericValue,
HTuple *BarcodeFound, HTuple *BarCodeElements, HTuple *Orientation )
const
```

```
HRegion HBarCode1D::Find1dBarCode ( const HImage &Image,
const HTuple &GenericName, const HTuple &GenericValue,
HTuple *BarcodeFound, HTuple *BarCodeElements, HTuple *Orientation )
const
```

`Image` (input_object) . . . . . . . . . . . . . . . . . . . . . . . . . . image ⇝ *Hobject: HImage* ( byte / uint2 )

`CodeRegion` (output_object) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . region ⇝ *Hobject * : HRegion*

`BarCodeDescr` (input_control) . . . . barcode_1d-array ⇝ *HTuple.const char ** / long / double

`GenericName` (input_control) . . . . . . . . . . . . . . . . . . . . . . . .string(-array) ⇝ *HTuple.const char **

`GenericValue` (input_control) . . . . . . . . . . . . . . . . . . number(-array) ⇝ *HTuple.double* / long

`BarcodeFound` (output_control) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . integer ⇝ *HTuple.long **

`BarCodeElements` (output_control) . . . . . . . . . . . . . . . . . . . .number-array ⇝ *HTuple.double **

`Orientation` (output_control) . . . . . . . . . . . . . . . . . . . . . . . . . . . .angle.rad ⇝ *HTuple.double **

Figure 2.2: The head and parts of the parameter section of the reference manual entry for `find_1d_bar_code`.

```
HImage      original_image("monkey"), smoothed_image;
smoothed_image = original_image.MeanImage(11, 11);
```

In contrast to input parameters, output parameters are always modified, thus they must be passed *by reference*. Note that operators expect a pointer to **an already declared variable**! For example, when calling the operator `Find1dBarCode` as in the following lines of code, variables of the class `HTuple` are declared before passing the corresponding pointers using the operator `&`.

```
HImage          image("barcode/ean13/ean1301");
HBarCode1D      barcode("EAN 13", 13, 13);
HRegion         code_region;
HTuple          isfound, elements;

code_region = barcode.Find1dBarCode(image, HTuple(), HTuple(),
                                    &isfound, &elements, (HTuple*) _);
```

The above example shows two other interesting aspects of output parameters: When calling operators via classes, one output parameter becomes the return value (see section 2.1.2 for more details); in the example, `Find1dBarCode` returns the bar code region. Secondly, a special variable called `_` was specified for the last parameter of `Find1dBarCode`. This variable is pre-declared; it tells the operator that it does not need to output the corresponding parameter.

```
Herror ::info_framegrabber ( const HTuple &Name, const HTuple &Query,
char *Information, char *ValueList )
```

```
Herror ::info_framegrabber ( const HTuple &Name, const HTuple &Query,
HTuple *Information, HTuple *ValueList )
```

`Name` (input_control) . . . . . . . . . . . . . . . . string $\rightsquigarrow$ *HTuple.const char **

`Query` (input_control) . . . . . . . . . . . . . . . . string $\rightsquigarrow$ *HTuple.const char **

`Information` (output_control) . . . . . . . . . . . . . string $\rightsquigarrow$ *(HTuple.) char **

`ValueList` (output_control) . . . . . . . . . . string(-array) $\rightsquigarrow$ *(HTuple.) char ** / long * / double *

Figure 2.3: The head and parts of the parameter section of the reference manual entry for `info_framegrabber`.

Many HALCON operators accept more than one value for certain parameters. For example, you can call the operator `MeanImage` with an array of images (see figure 2.1); then, an array of smoothed images is returned. This is called the *tuple mode*; see section 2.1.3 for more information.

! Please note that **output parameters of the type string need special attention**: First of all, you must allocate memory for them yourself, e.g., by declaring them as character arrays; secondly, you don't pass them by reference, because such parameters are pointers already. In the following example code, the operator `info_framegrabber` (see also figure 2.3) is called with two output string parameters to query the currently installed PicPort frame grabber board:

```
const char   *FGName = "PicPort";
char         sInfo[MAX_STRING], sValue[MAX_STRING];

::info_framegrabber(FGName, "info_boards", sInfo, sValue);
```

Note that it isn't necessary to allocate memory for output string parameters in the already mentioned *tuple mode*, i.e., when using instances of the class `HTuple` instead of "plain" strings (also see sections 2.1.3 and 3.2.2):

```
HTuple       tInfo, tValues;

::info_framegrabber(FGName, "camera_types", &tInfo, &tValues);
```

## 2.1.2 Calling Operators via Classes

As already described in the previous section, the HALCON/C++ reference manual shows via which classes an operator can be called. For example, `Find1dBarCode` can be called via objects of the class `HImage` or `HBarCode1d` (see figure 2.2). In both cases, the corresponding input parameter (`Image` or `BarCodeDescr`, respectively) does not appear within the brackets anymore as it is replaced by the calling instance of the class (`this`).

There is a further difference to the procedural operator signature: The first output parameter (in the example the bar code region `CodeRegion`) also disappears from within the brackets and

```
HImage          image("barcode/ean13/ean1301");
HBarCode1D      barcode("EAN 13", 13, 13);
HRegion         code_region;
HTuple          isfound, elements;

code_region = barcode.Find1dBarCode(image, HTuple(), HTuple(),
                                     &isfound, &elements, (HTuple*) _);
```

```
code_region = image.Find1dBarCode(barcode.GetHandle(), HTuple(), HTuple(),
                                   &isfound, &elements, (HTuple*) _);
```

```
Hobject         image;
HTuple          barcode;
Hobject         code_region;
HTuple          isfound, elements;

::read_image(&image, "barcode/ean13/ean1301");
::gen_1d_bar_code_descr("EAN 13", 13, 13, &barcode);
::find_1d_bar_code(image, &code_region, barcode, HTuple(), HTuple(),
                   &isfound, &elements, (HTuple*) _);
```

Figure 2.4: Using `Find1dBarCode` via `HBarCode1d`, via `HImage`, or in the procedural approach.

becomes the return value instead of the error code (more about error handling can be found in section 2.2).

Figure 2.4 depicts code examples for the three ways to call `Find1dBarCode`. When comparing the object-oriented and the procedural approach, you can see that the calls to the operators `read_image` and `gen_1d_bar_code_descr` are replaced by special constructors for the classes `HImage` and `HBarCode1d`, respectively. This topic is discussed in more detail below.

Please note that the two object-oriented methods seem to be "asymmetric": If you call `Find1dBarCode` via `HImage`, the reference manual seems to suggest that you must pass the *handle* instead of an instance of `HBarCode1d`. In fact, you can pass both a handle and a class instance, because the latter is automatically "casted" into a handle; the signature was not changed to keep the HALCON/C++ interface backward compatible as far as possible.

#### 2.1.2.1 Constructors

As can be seen in figure 2.4, the HALCON/C++ parameter classes provide additional constructors which are based on suitable HALCON operators. The constructors for `HImage` and `HBarCode1d` used in the example are based on `read_image` and `gen_1d_bar_code_descr`, respectively.

Please note that in the current HALCON version constructors are provided inconsistently for the different classes. Below we take a brief look at the most important classes. A complete and up-to-date list of available constructors can be found in the corresponding header files in `%HALCONROOT%\include\cpp`.

- **Images:**
  The class `HImage` provides constructors based on the operators `read_image`, `gen_image1`, `gen_image1_extern`, and `gen_image_const`.

Please **beware of the following pitfall** when using the operators themselves via HImage: Contrary to intuition, the operators do not modify the instance they are called from; instead, the created image is the return value of the operator! Thus, after the following code the image is still uninitialized:

```
HImage          image;
image.ReadImage("barcode/ean13/ean1301");            // incorrect
```

The correct way to call ReadImage is as follows:

```
image = HImage::ReadImage("barcode/ean13/ean1301");  // correct
```

Note that this pitfall concerns all operators where HImage appears as an output parameter, e.g., GrabImage. More information about HImage can be found in section 3.1.2.

- **Regions:**
  The class HRegion provides constructors based on operators like gen_rectangle2 or gen_circle. However, instead of the parameters of these operators, the constructors expect instances of auxiliary classes like HRectangle2 or HCircle (see section 3.3 for more information about these classes).

  Please note that HRegion **presents the same pitfall** as HImage, i.e., operators like GenRectangle2 do not modify the calling instance of HRegion but return the created region! More information about HRegion can be found in section 3.1.1.

- **XLDs:**
  The classes for XLDs (HXLD, HXLDCont, etc., see section 3.1.3 for more information) do not provide constructors based on operators.

- **Windows:**
  The class HWindow provides constructors based on the operators open_window and new_extern_window. Note that the former is realized with default values for all parameters, thus becoming the default constructor, i.e., all window instances are already opened upon construction!

  Of course, you can close a window using CloseWindow and then open it again using OpenWindow. In contrast to the iconic parameter classes, you can call the "constructor-like" operator OpenWindow via an instance of HWindow in the intuitive way, i.e., the calling instance is modified; in addition the corresponding handle is returned. HWindow is described in more detail in section 3.2.3.1.

- **Other Handle Classes:**
  The other classes encapsulating handles, e.g., HBarCode1d or HFramegrabber, provide constructors in a systematic way: If a class appears as an output parameter in an operator, there automatically exists a constructor based on this operator. Thus, instances of HBarCode1d can be constructed based on gen_1d_bar_code_descr as shown in figure 2.4, instances of HShapeModel based on create_shape_model, instances of HFramegrabber based on open_framegrabber and so on.

  In contrast to the iconic parameter classes, handle classes allow to call constructor-like operators via instances of the class in the intuitive way, i.e., the calling instance is modified. For example, you can create an instance of HBarCode1d with the default constructor and then initialize it using Gen1dBarCodeDescr as follows:

```
HBarCode1D      barcode;
barcode.Gen1dBarCodeDescr("EAN 13", 13, 13);
```

If the instance was already initialized, the corresponding data structures are automatically destroyed before constructing and initializing them anew (see also section 2.1.2.2). The handle classes are described in more detail in section 3.2.3.2.

#### 2.1.2.2 Destructors

All HALCON/C++ classes provide default destructors which automatically free the corresponding memory. For some classes, the destructors are based on suitable operators:

- **Windows:**
  The default destructor of the class `HWindow` closes the window based on `close_window`. Note that the operator itself is no destructor, i.e., you can close a window and then open it again using `OpenWindow`.

- **Other Handle Classes:**
  The default destructors of the other classes encapsulating handles, e.g., `HShapeModel` or `HFramegrabber`, apply operators like `clear_shape_model` or `close_framegrabber`, respectively. In contrast to `close_window`, these operators cannot be called via instances of the class, as can be seen in the corresponding reference manual entries; the same holds for operators like `clear_all_shape_models`. In fact, there is no need to call these operators as you can initialize instances anew as described in section 2.1.2.1.

  Please note that you must not use operators like `clear_shape_model`, `clear_all_shape_models`, or `close_framegrabber` together with instances of the corresponding handle classes!

### 2.1.3 The Tuple Mode

As already mentioned in section 2.1.1, many HALCON operators can be called in the so-called *tuple mode*. In this mode, you can, e.g., apply an operator to multiple images or regions with a single call. The standard case, e.g., calling the operator with a single image, is called the *simple mode*. Whether or not an operator supports the tuple mode can be checked in the reference manual. For example, take a look at figure 2.5, which shows an extract of the reference manual entry for the operator `char_threshold`: In the parameter section, the parameter `Image` is described as an `image(-array)`; this signals that you can apply the operator to multiple images at once.

If you call `char_threshold` with multiple images, i.e., with an image tuple, the output parameters automatically become tuples as well. Consequently, the parameters `Characters` and `Threshold` are described as `region(-array)` and `integer(-array)`, respectively.

The head section of the reference entry in figure 2.5 shows how simple and tuple mode are reflected in the operator's signatures. In the procedural approach, the simple and tuple mode methods of calling `char_threshold` differ only in the type of the output parameter `Threshold`: a pointer to a `long` or to a `HTuple` of `long` values, respectively. Note that the class `HTuple` can also contain arrays (tuples) of control parameters of mixed type; please refer to section 3.2.2 for more information about this class. In contrast to the control parameters, the iconic parameters remain instances of the class `Hobject` in both modes, as this class can contain both single objects and object arrays (see also section 3.1.4).

```
Herror ::char_threshold ( Hobject Image, Hobject HistoRegion,
Hobject *Characters, const HTuple &Sigma, const HTuple &Percent,
long *Threshold )

Herror ::char_threshold ( Hobject Image, Hobject HistoRegion,
Hobject *Characters, const HTuple &Sigma, const HTuple &Percent,
HTuple *Threshold )

HRegion HImage::CharThreshold ( const HRegion &HistoRegion,
const HTuple &Sigma, const HTuple &Percent, long *Threshold ) const

HRegionArray HImageArray::CharThreshold ( const HRegion &HistoRegion,
const HTuple &Sigma, const HTuple &Percent, HTuple *Threshold ) const
```

`Image` (input_object) . . . . . . . . . . . . . . . . . . . image(-array) $\rightsquigarrow$ *Hobject: HImage(Array)* ( byte )

`HistoRegion` (input_object) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . region $\rightsquigarrow$ *Hobject: HRegion*

`Characters` (output_object) . . . . . . . . . . . . . . . region(-array) $\rightsquigarrow$ *Hobject * : HRegion(Array)*

`Sigma` (input_control) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . number $\rightsquigarrow$ *HTuple.double*

`Percent` (input_control) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . number $\rightsquigarrow$ *HTuple.double* / long

`Threshold` (output_control) . . . . . . . . . . . . . . . . . . . . . . . . . . integer(-array) $\rightsquigarrow$ *(HTuple.) long **

Figure 2.5: The head and parts of the parameter section of the reference manual entry for `CharThreshold`.

In the object-oriented approach, simple mode and tuple mode methods use different classes for the iconic parameters: `HImage` and `HRegion` vs. `HImageArray` and `HRegionArray` (see sections 3.1.1 and 3.1.2 for more information about these classes). As in the procedural approach, control parameters can be of a basic type (simple mode only) or instances of `HTuple` (simple and tuple mode).

After this rather theoretic introduction, let's take a look at example code. In figure 2.6, `char_threshold` is applied in simple mode, i.e., to a single image, in figure 2.7 to two images at once. Both examples are realized both in the object-oriented and in the procedural approach. The examples highlight some interesting points:

- **Creation and initialization of iconic arrays:**
  In the object-oriented approach, the image array can be constructed very easily by *assigning* the individual images to certain positions in the array using the well-known array operator `[]`. In the procedural approach, you must explicitely create an empty object using `gen_empty_obj` and then add the images via `concat_obj`.

- **Access to iconic objects:**
  As expected, in the object-oriented approach, the individual images and regions are accessed via the array operator `[]`; the number of objects in an array can be queried via the method `Num()`. In the procedural approach, objects must be selected explicitely using the operator `select_obj`; the number of objects can be queried via `count_obj`.

- **Polymorphism of `Hobject`:** (part I)
  As already noted, instances of `Hobject` can be used both in simple and in tuple mode. In

```
HImage          image("alpha1");
HRegion         region;
long            threshold;

region = image.CharThreshold(image.GetDomain(), 2, 95, &threshold);
image.Display(window);
region.Display(window);
cout << "Threshold for 'alpha1': " << threshold;
```

```
Hobject         image;
Hobject         region;
long            num;
long            threshold;

::read_image(&image, "alpha1");
::char_threshold(image, image, &region, 2, 95, &threshold);
::disp_obj(image, window);
::disp_obj(region, window);
cout << "Threshold for 'alpha1': " << threshold;
```

Figure 2.6: Using `CharThreshold` in simple mode, via `HImage`, or in the procedural approach (declaration and opening of window omitted).

contrast, you must use different classes when switching from simple to tuple mode in the object-oriented approach.

- **Polymorphism of `Hobject`:** (part II)
  The class `Hobject` is used for all types of iconic objects. What's more, image objects can be used for parameters expecting a region, as in the call to `char_threshold` in the examples; in this case, the *domain* of the image, i.e., the region in which the pixels are "valid", is extracted automatically. In the object-oriented approach, you must extract the domain explicitely via the operator `GetDomain`.

- **Array (tuple) indices:**
  Object-oriented iconic arrays start with the index 0, the same is true for `HTuple`. In contrast, `Hobject` arrays start with the index 1!!

Most of the time you will call operators in tuple mode without noticing: As soon as you divide a region into connected components via the operator `Connection`, you end up with a `HRegionArray`– thus, any subsequent processing, e.g., morphological operations like `DilationCircle` or the calculation of the region's position using `AreaCenter` is automatically performed on all regions in the array, i.e., in tuple mode. Thus, the tuple mode is a simple mode after all!

## 2.2 Error Handling

In case of a runtime error, HALCON/C++ by default prints a corresponding error message and terminates the program. In some applications, however, it might be useful to slacken this rule for certain errors. For example, if an application allows the user to specify an image file to read interactively, it would be inconvenient if the application terminates because the user misspelled

```
HImageArray    images;
HRegionArray   regions;
HTuple         thresholds;

for (int i=1; i<=2; i++)
{
  images[i-1] = HImage::ReadImage(HTuple("alpha") + i);
}

regions = images.CharThreshold(images[0].GetDomain(), 2, 95, &thresholds);

for (int i=0; i<images.Num(); i++)
{
  images[i].Display(window);
  regions[i].Display(window);
  cout << "Threshold for 'alpha" << i+1 << "': " << thresholds[i].L();
}
```

```
Hobject        images, image;
Hobject        regions, region;
long           num;
HTuple         thresholds;

::gen_empty_obj(&images);

for (int i=1; i<=2; i++)
{
  ::read_image(&image, HTuple("alpha") + i);
  ::concat_obj(images, image, &images);
}

::char_threshold(images, image, &regions, 2, 95, &thresholds);
::count_obj(images, &num);

for (int i=0; i<num; i++)
{
  ::select_obj(images, &image, i+1);
  ::disp_obj(image, window);
  ::select_obj(regions, &region, i+1);
  ::disp_obj(region, window);
  cout << "Threshold for 'alpha" << i+1 << "': " << thresholds[i].L();
}
```

Figure 2.7: Using `CharThreshold` in tuple mode, via `HImageArray`, or in the procedural approach (declaration and opening of window omitted).

the file name. Therefore, HALCON/C++ allows to integrate your own error handling. How to do this in the object-oriented and in the procedural approach is described in the following sections. Please note that you cannot mix object-oriented and procedural error handling.

```
class LIntExport HException {
public:
  HException(const HException &except);
  HException(const char *f, long l, const char *p, Herror e, const char *m);
  HException(const char *f, long l, const char *p, const char *m);
  HException(const char *f, long l, const char *p, Herror e);

  static Handler InstallHHandler(Handler proc);
  void           PrintException(void);

  static Handler handler;   /* handler in use */
  long           line;      /* line number where error occured */
  const char     *file;     /* file name where error occured */
  const char     *proc;     /* Name of procedure/operator */
  Herror         err;       /* error number */
  char           *message;  /* error text */
};
```

Figure 2.8: Part of the declaration of the class `HException`.

## 2.2.1 Object-Oriented Approach

If a runtime error occurs in an object-oriented operator call, an instance of the class `HException` is created (see figure 2.8 for the declaration of the class). This instance contains all information concerning the error. The important members of an exception are:

| | |
|---|---|
| `line`: | Number of the program line in which the error occurred |
| `file`: | Name of the file in which the error occurred |
| `proc`: | Name of the actual HALCON operator |
| `err`: | Number of the error, see below |
| `message`: | Error text |

After the generation, the instance of `HException` is passed to a so-called *exception handler*. HALCON's default exception handler prints the corresponding error message and terminates the program.

As an alternative, you can implement and use your own exception handler. In order to act as a HALCON exception handler, a procedure must have the following signature:

```
typedef void (*Handler)(const HException &exception);
```

You "install" your exception handler procedure via `HException`'s class method `InstallHHandler` (see figure 2.8). In case of a runtime error, HALCON then calls your procedure, passing the instance of the actual exception as a parameter.

The following example shows how to use a user-specific exception handler together with the standard C++ exception handling mechanism (`try...catch`). The corresponding program `example_errorhandling.cpp` can be found in the subdirectory `%HALCONROOT%\examples\cpp`. It realizes the application mentioned above: You can type in image files to load; if a file does not exist, the program prints a corresponding message but continues nevertheless.

At the beginning of the program, a user-specific exception handler is installed with the folloing line:

```
  HException::InstallHHandler(&MyHalconExceptionHandler);
```

The installed procedure simply hands the exception object to the C++ exception handling via `throw`:

```
void MyHalconExceptionHandler(const HException& except)
{
  throw except;
}
```

The call to `ReadImage` is then encapsulated by a `try` block; a possibly ensuing exception is then evaluated in a corresponding `catch` block:

```
  try
  {
    image = HImage::ReadImage(filename);
  }
  catch (HException &except)
  {
    error_num = except.err;
    if (error_num == H_MSG_FAIL)
      cout << "image not found!" << endl;
    else
      cout << endl << except.message << endl;
  }
```

### 2.2.2 Procedural Approach

As can be seen in the extracts of the reference manual in section 2.1, in the procedural approach operators return a value of the type `Herror`. This value can fall into two categories: messages `H_MSG_*` and errors `H_ERR_*`. There are four different messages:

| | |
|---|---|
| `H_MSG_TRUE`: | The operator terminated without an error and the result value is the boolean value true. |
| `H_MSG_FALSE`: | The operator terminated without an error and the result value is the boolean value false. |
| `H_MSG_VOID`: | The operator terminated without an error and the result value void is returned. |
| `H_MSG_FAIL`: | The operator terminated without an error and the result value means the operator has not performed successfully. This means, e.g. an operator is not responsible or a specific situation has not occurred. |

Typically, HALCON operators return the message `H_MSG_TRUE` if no error occurs.

In case of an error, HALCON by default prints the corresponding error message and terminates the program. You can deactivate (and reactivate) this reaction by calling the operator `set_check`. The following example code checks whether a file could be opened successfully; in case of an error, it prints the corresponding error message, which can be determined with the operator `get_error_text`.

```
Herror          error_num;
char            message[MAX_STRING];
long            file;

::set_check("~give_error");
error_num = ::open_file("not_existing_file", "input", &file);
::set_check("give_error");

if (error_num != H_MSG_TRUE)
{
  ::get_error_text(error_num, message);
  cout << "HALCON error " << error_num << ": " << message;
}
```

## 2.3 Memory Management

All of HALCON's classes, i.e., not only HImage, HRegion, HTuple, HFramegrabber etc., but also the class Hobject used when calling operators in the procedural approach, release their allocated memory automatically in their default destructor (see also section 2.1.2.2). Furthermore, when constructing instances anew, e.g., by calling Gen1dBarCodeDescr via an already initialized instance as mentioned in section 2.1.2.1, the already allocated memory is automatically released before allocating it anew. Thus, there is no need to call the operator clear_obj in HALCON/C++; what's more, if you do use it HALCON will complain about already released memory.

The only occasion for explicit memory management on your part is when using handles in the procedural approach: The memory allocated when creating a handle, e.g., with open_framegrabber, is only released when calling the "complementary" operator, in the example close_framegrabber — or at the end of the program.

## 2.4 How to Combine Procedural and Object-Oriented Code

As already noted, we recommend to use the object-oriented approach wherever possible. However, there are some reasons for using the procedural approach, e.g., if you want to quickly integrate code that is exported by HDevelop, which can only create procedural code. Besides, currently some operators are only available in procedural form, e.g., operators creating affine transformations like vector_to_rigid.

The least trouble is caused by the basic control parameters as both approaches use the elementary types long etc. and the class HTuple. Iconic parameters and handles can be converted as follows:

- **Converting Hobject into iconic parameter classes**

```
Hobject         p_image;
::read_image(&p_image, "barcode/ean13/ean1301");

HImage          o_image(p_image);
```

Iconic parameters can be converted from Hobject to, e.g., HImage simply by calling the constructor with the procedural variable as a parameter.

- **Converting handles into handle classes**

```
HTuple          p_barcode;
::gen_1d_bar_code_descr("EAN 13", 13, 13, &p_barcode);

HBarCode1D      o_barcode;
o_barcode.SetHandle(p_barcode);
o_code_region = o_barcode.Find1dBarCode(o_image, HTuple(), HTuple(),
                                        &isfound, &elements,
                                        (HTuple*) _);
```

Handles cannot be converted directly via a constructor; instead, you call the method SetHandle() with the procedural handle as a parameter.

- **Converting handle classes into handles**

```
p_barcode = o_barcode.GetHandle();
::decode_1d_bar_code(elements, p_barcode, (HTuple*) _, &result,
                     (HTuple*) _);
```

Similarly, a handle can be extracted from the corresponding class via the method GetHandle(). You can even omit the method, as the handle classes provide cast operators which convert them automatically into handles.

```
p_barcode = o_barcode;
```

- **Converting iconic parameter classes into** Hobject

```
Hobject         p_code_region = o_code_region.Id();
```

Iconic parameters can be converted from classes like HRegion back into Hobject via the method Id(). In contrast to the handle classes no cast operator is provided.

- **Converting** HWindow **into a window handle**

```
long            p_window;
::open_window(0, 0, width/2, height/2, 0, "visible", "", &p_window);

HWindow         o_window(0, 0, 100, 100, 0, "visible", "");

p_window = o_window.WindowHandle();
::disp_obj(p_code_region, p_window);
```

In contrast to other handles, procedural window handles cannot be converted into instances of the class HWindow! However, you can extract the handle from an instance of HWindow via the method WindowHandle().

As already remarked in section 2.1.2.2, you must not use operators like clear_shape_model, clear_all_shape_models, or close_framegrabber together with instances of the corresponding handle classes!

# Chapter 3

# The HALCON Parameter Classes

## 3.1 Iconic Objects

The base class of the iconic parameter classes in HALCON/C++ is the (abstract) class `HObject` which manages entries in the database, i.e., the copying or releasing of objects. The entries themselves are represented by the class `Hobject` (see also section 3.1.4). The classes `HObject` and `Hobject` can contain all types of iconic objects. This has the advantage that important methods like `Display()` can be applied to all iconic objects in the same manner.

Three classes are derived from the root class `HObject`:

- Class `HImage` for handling images.
- Class `HRegion` for handling regions.
- Class `HXLD` for handling polygons.

These classes are described in more detail below.

### 3.1.1 Regions

A region is a set of coordinates in the image plane. Such a region does not need to be connected and it may contain holes. A region can be larger than the actual image format. Regions are represented by the so-called runlength coding in HALCON. The class `HRegion` represents a region in HALCON/C++. Besides those operators that can be called via `HRegion` (see also section 2.1.2), `HRegion` provides the following member functions:

- `HRegion(void)`
  Default constructor. It creates an empty region, i.e., the area of this region is zero. Not all operators can handle the empty region as input, e.g. some shape property operators.
- `HRegion(const HDChord &line)`
  Constructing a region from a chord. A chord is a horizontal line.
- `HRegion(const HDPoint2D &point)`
  Constructing a region from a discrete 2-dimensional point.

- `HRegion(const HRectangle1 &rect)`
  Constructing a region from a rectangle parallel to the coordinate axis. The coordinates do not need to be discrete.

- `HRegion(const HRectangle2 &rect)`
  Constructing a region from an arbitrarily oriented rectangle. The coordinates do not need to be discrete.

- `HRegion(const HCircle &circle)`
  Constructing a region from a circle. The radius and center do not need to be discrete.

- `HRegion(const HEllipse &ellipse)`
  Constructing a region from an arbitrarily oriented ellipse. The radii and center do not need to be discrete.

- `HRegion(const char *file)`
  Constructing a region by reading the representation from file. This file can be generated by the member function `WriteRegion`.

- `HRegion(const HRegion &reg)`
  Copy constructor.

- `HRegion &operator = (const HRegion &reg)`
  Assignment operator.

- `~HRegion(void)`
  Destructor. In contrast to the primitive class (`Hobject`) this class handles the release of memory.

- `void Display(const HWindow &w) const`
  Output of the region in a window.

- `HRegion  operator * (double scale) const`
  Zooming the region by an arbitrary factor. The center of scaling is the origin $(0, 0)$.

- `HRegion  operator >> (double radius) const`
  `HRegion &operator >>= (double radius)`
  Erosion of the region with a circle of radius `radius`, see reference manual entry of `erosion_circle`.

- `HRegion  operator << (double radius) const`
  `HRegion &operator <<= (double radius)`
  Dilation of the region with a circle of radius `radius`, see reference manual entry `dilation_circle`.

- `HRegion  operator + (const HDPoint2D &point) const`
  `HRegion &operator += (const HDPoint2D &point)`
  Translating the region by a 2-dimensional point.

- `HRegion &operator ++ (void)`
  Dilation of the region with a cross containing five points.

- `HRegion  operator + (const HRegion &reg) const`
  `HRegion &operator += (const HRegion &reg)`
  Minkowsky addition of the region with another region, see reference manual entry of `minkowski_add1`.

- `HRegion  operator - (const HRegion &reg) const`
  `HRegion &operator -= (const HRegion &reg)`
  Minkowsky subtraction of the region with another region, see reference manual entry of `minkowski_sub1`.

- `HRegion &operator -- (void)`
  Erosion of the region with a cross containing five points.

- `HRegion  operator ~ (void) const`
  Complement of the region, see reference manual entry of `complement`.

- `HRegion  operator ! (void) const`
  Transpose the region at the origin, see reference manual entry of `transpose_region`.

- `HRegion  operator & (const HRegion &reg) const`
  `HRegion &operator &= (const HRegion &reg)`
  Intersection of the region with another region, see reference manual entry of `intersection`.

- `HRegion  operator | (const HRegion &reg) const`
  `HRegion &operator |= (const HRegion &reg)`
  Union of the region with another region, see reference manual entry of `union2`.

- `HRegion  operator / (const HRegion &reg) const`
  `HRegion &operator /= (const HRegion &reg)`
  Subtract another region from the region, see reference manual entry of `difference`.

- `HBool operator == (const HRegion &reg) const`
  Boolean test if two regions are identical, see reference manual entry of `test_equal_region`.

- `HBool operator >= (const HRegion &reg) const`
  `HBool operator > (const HRegion &reg) const`
  `HBool operator <= (const HRegion &reg) const`
  `HBool operator < (const HRegion &reg) const`
  Boolean test if another region is included in the region by using the subset of the corresponding coordinates.

- `double Phi(void) const`
  Orientation of the region by using the angle of the equivalent ellipse, see reference manual entry of `elliptic_axis`.

- `double Ra(void) const`
  Length of the major axis of the equivalent ellipse of the region, see reference manual entry of `elliptic_axis`.

- `double Rb(void) const`
  Length of the minor axis of the equivalent ellipse of the region, see reference manual entry of `elliptic_axis`.

- `long Area(void) const`
  Area of the region, i.e., number of pixels, see reference manual entry of `area_center`.

- `double X(void) const`
  `double Y(void) const`

Center point of the region, see reference manual entry of `area_center`.

- `double Contlength(void) const`
  Length of the contour of the region, see reference manual entry of `contlength`.

- `double Compactness(void) const`
  Compactness of the actual region, see reference manual entry of `compactness`.

- `double Anisometry(void) const`
  `double Bulkiness(void) const`
  `double StructureFactor(void) const`
  Shape factors, see reference manual entry of `eccentricity`.

- `double M11(void) const`
  `double M20(void) const`
  `double M02(void) const`
  `double Ia(void) const`
  `double Ib(void) const`
  Moments of the region, see reference manual entry of `moments_region_2nd`.

- `HRectangle1 SmallestRectangle1(void) const`
  Smallest surrounding rectangle parallel to the coordinate axis, see reference manual entry of `smallest_rectangle1`.

- `HBool In(const HDPoint2D &p) const`
  Boolean test if a point is inside a region, see reference manual entry of `test_region_point`.

- `HBool IsEmpty(void) const;`
  Boolean test if the region is empty, i.e., the area of the region is zero.

A program shows the power of the class `HRegion`, see figure 3.1.

First, an aerial image (`mreut.tiff`) is read from a file. All pixels with a gray value $\geq 190$ are selected. This results in one region (`region`).

This region is transformed by the next steps: All holes in the region are filled (`FillUp`), small parts of the region are eliminated by two morphological operations, first an erosion, a kind of shrinking the region, followed by a dilation, a kind of enlarging the region. The last step is the zooming of the region. For that the region is first shifted by a translation vector $(-100, -150)$ to the upper left corner and then zoomed by the factor two. Figure 3.2 shows the input image and the result of the opening operation.

#### 3.1.1.1 Region Arrays

The class `HRegionArray` serves as container class for regions. Besides those operators that can be called via `HRegionArray` (see also section 2.1.2), `HRegionArray` provides the following member functions:

- `HRegionArray(void)`
  Constructor for an empty array (`Num()` is 0).

- `HRegionArray(const HRegion &reg)`
  Constructor with a single region.

```
#include "HalconCpp.h"
#include "iostream.h"

main ()
{
  HImage     image("mreut");              // Reading an aerial image
  HRegion    region = image >= 190;       // Calculating a threshold
  HWindow    w;                           // Display window
  w.SetColor("red");                      // Set color for regions
  region.Display(w);                      // Display the region
  HRegion    filled = region.FillUp();    // Fill holes in region
  filled.Display(w);                      // Display the region
  // Opening: erosion followed by a dilation with a circle mask
  HRegion    open =  (filled >> 4.5) << 4.5;
  w.SetColor("green");                    // Set color for regions
  open.Display(w);                        // Display the region
  HDPoint2D  trans(-100,-150);            // Vector for translation
  HRegion    moved  = open + trans;       // Translation
  HRegion    zoomed = moved * 2.0;        // Zooming the region
}
```

Figure 3.1: Sample program for the application of the class `HRegion` .

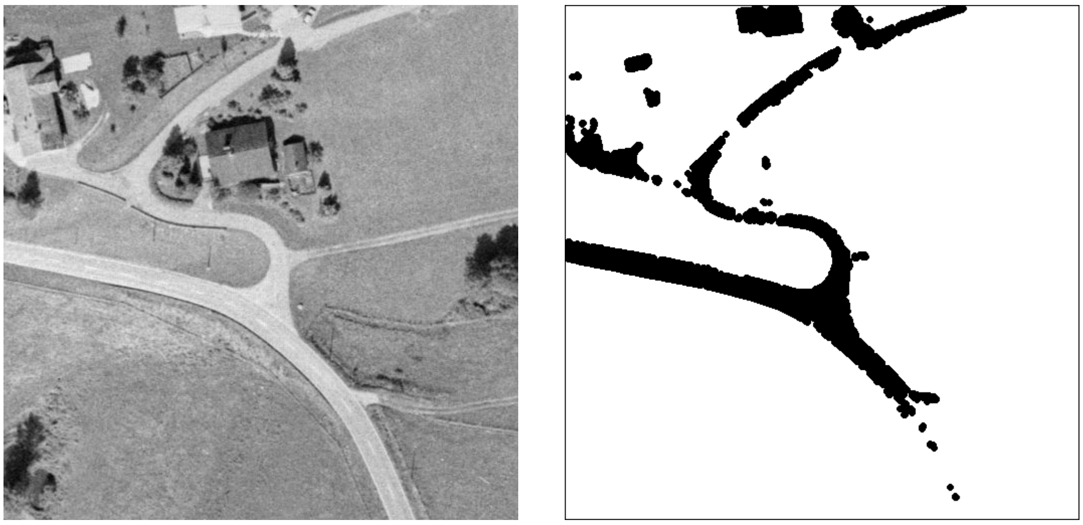

Figure 3.2: On the left the input image (`mreut.tiff`), and on the right the region after the opening (`open`).

- `HRegionArray(const HRegionArray &arr)`
  Copy constructor.
- `~HRegionArray(void)`
  Destructor.
- `HRegionArray &operator = (const HRegionArray &arr)`
  Assignment operator.

- `long Num(void)`
  Number of regions in the array, largest index is `Num()` $- 1$.
- `HRegion const &operator [] (long index) const`
  Reading the element $i$ of the array. The index is in the range $0 \ldots$ `Num()` $- 1$.
- `HRegion &operator [] (long index)`
  Assigning a region to the element $i$ of the array. The index `index` can be $\geq$ `Num()`.
- `HRegionArray operator () (long min, long max) const`
  Selecting a subset between the lower `min` and upper `max` index.
- `HRegionArray &Append(const HRegion &reg)`
  Appending another region to the region array.
- `HRegionArray &Append(const HRegionArray &reg)`
  Appending another region array to the region array.
- `void Display(const HWindow &w) const`
  Display the regions of the array in a window.
- `HRegionArray operator << (double radius) const`
  Applying a dilation to all regions using a circular mask, see reference manual entry of `dilation_circle`.
- `HRegionArray operator >> (double radius) const`
  Applying an erosion to all regions using a circular mask, see reference manual entry of `erosion_circle`.
- `HRegionArray operator + (const HRegion &reg) const`
  Applying the Minkowsky addition to all regions using another region as mask, see reference manual entry of `minkowski_add1`.
- `HRegionArray operator - (const HRegion &reg) const`
  Applying the Minkowsky subtraction to all regions using another region as mask, see reference manual entry of `minkowski_sub1`.
- `HRegionArray operator ~ (void) const`
  Applying the complement operator to each region of the array, see reference manual entry of `complement`.
- `HRegionArray operator & (const HRegionArray &reg) const`
  Intersection of each region of the actual array with the union of `reg`, see reference manual entry of `intersection`.
- `HRegionArray operator | (const HRegionArray &reg) const`
  Union of each region in the actual array with the union of `reg`, see reference manual entry of `union2`.
- `HRegionArray operator / (const HRegionArray &reg) const`
  Difference of each region in the actual array with the union of `reg`, see reference manual entry of `difference`.

Most HALCON operators expecting a region for an input parameter accept an instance of `HRegionArray`, e.g. `Union1`, `Intersection`, `Difference`, etc. The constructor instantiating the region array `HRegionArray` from a single region `HRegion` makes it possible to handle operators expecting a single region: Without changing the data structure a HRegionArray can be used as input parameter even in the case of a single region.

Figure 3.3 shows a short example how to use the class HRegionArray.

```
#include "HalconCpp.h"
#include "iostream.h"

main ()
{
  HImage        image("control_unit");       // Reading an image from file
  // Segmentation by regiongrowing
  HRegionArray  regs = image.Regiongrowing(1,1,4,100);
  HWindow       w;                           // Display window
  w.SetColored(12);                          // Set colors for regions
  regs.Display(w);                           // Display the regions
  HRegionArray  rect;                        // New array
  for (long i = 0; i < regs.Num(); i++)      // For all regions in array
  { // Test size and shape of each region
    if ((regs[i].Area() > 1000) && (regs[i].Compactness() < 1.5))
      rect.Append(regs[i]);                  // If test true, append region
  }
  image.Display(w);                          // Display the image
  rect.Display(w);                           // Display resulting regions
}
```

Figure 3.3: Sample program for use of the class HRegionArray .

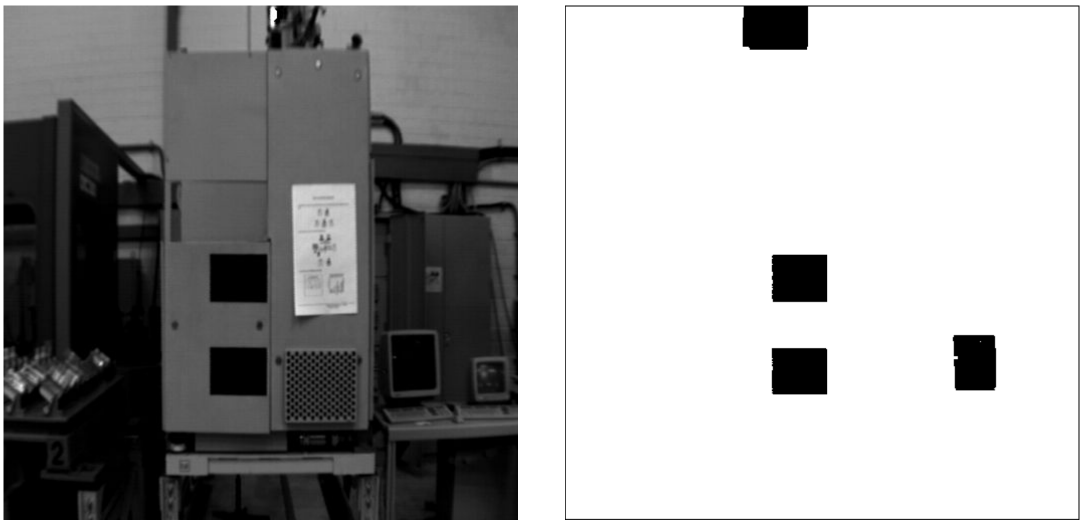

Figure 3.4: On the left side the input image (control_unit.tiff), and on the right side the selected rectangles.

The first step is to read an image. In this case it shows a control unit in a manufacturing environment, see figure 3.4 on the left side. By applying a regiongrowing algorithm from the HALCON library the image is segmented into regions. Each region inside the resulting region array `regs` is now selected according to its size and its compactness. Each region of a size larger than 1000 pixels and of a compactness value smaller than $1.5$ is appended to the region array `rect`. After the processing of the `for` loop only the regions showing on the right side of figure 3.4 are left.

## 3.1.2 Images

There is more to HALCON images than just a matrix of pixels: In HALCON, this matrix is called a *channel*, and images may consist of one or more such channels. For example, gray value images consist of a single channel, color images of three channels. Channels can not only contain the standard 8 bit pixels (pixel type `byte`) used to represent gray value images, HALCON allows images to contain various other data, e.g. 16 bit integers (type `int2`) or 32 bit floating point numbers (type `real`) to represent derivatives. Besides the pixel information, each HALCON image also stores its so-called *domain* in form of a HALCON region. The domain can be interpreted as a region of interest, i.e., HALCON operators (with some exceptions) restrict their processing to this region.

### 3.1.2.1 Image Objects

The class `HImage` is the root class for all derived image classes. By using the class `HImage` all different pixel types can be handled in a unique way (polymorphism). The class `HImage` is not virtual, thus it can be instantiated. Besides those operators that can be called via `HRegion` (see also section 2.1.2), `HRegion` provides the following member functions:

- `HImage(void)`
  Default constructor, empty image.
- `HImage(const char *file)`
  Constructing an image by reading from a file, see reference manual entry of `read_image`.
- `HImage(int width, int height, const char *type)`
  Constructing an image of a defined size and a specific pixel type, see reference manual entry of `gen_image_const`.
- `HImage(void *ptr, int width, int height, const char *type)`
  Constructing an image of a defined size and a specific pixel type by copying memory, see reference manual entry of `gen_image1`.
- `HImage(const HImage &image)`
  Copy constructor.
- `virtual ~HImage(void)`
  Destructor.
- `HImage &operator = (const HImage &arr)`
  Assignment operator.
- `virtual const char *PixType(void) const`
  Return the pixel type of the image, see reference manual entry of `get_image_pointer1`.
- `int Width(void) const`
  Return the width of the image, see reference manual entry of `get_image_pointer1`.

- `int Height(void) const`
  Return the height of the image, see reference manual entry of `get_image_pointer1`.
- `HPixVal GetPixVal(int x, int y) const`
  Access a pixel value via the $(x, y)$ coordinates, see reference manual entry of `get_grayval`.
- `HPixVal GetPixVal(long k) const`
  Linear access of a pixel value.
- `virtual void SetPixVal(int x, int y, const HPixVal &val)`
  Set the pixel value via the $(x, y)$ coordinates, see reference manual entry of `set_grayval`.
- `virtual void SetPixVal(long k, const HPixVal &val)`
  Set the pixel value by linear access.
- `virtual void Display(const HWindow &w) const`
  Display an image in a window.
- `HImage  operator & (const HRegion &reg) const`
  Reduce the domain of an image, see reference manual entry of `reduce_domain`.
- `HImage  operator + (const HImage &add) const`
  Adding two images, see reference manual entry of `add_image`.
- `HImage  operator - (const HImage &sub) const`
  Subtracting two images, see reference manual entry of `sub_image`.
- `HImage  operator * (const HImage &mult) const`
  Multiplication of two images, see reference manual entry of `mult_image`.
- `HImage  operator - (void) const`
  Inverting the values of the image, see reference manual entry of `invert_image`.
- `HImage  operator + (double add) const`
  `HImage  operator - (double sub) const`
  `HImage  operator * (double mult) const`
  `HImage  operator / (double div) const`
  Arithmetic operators, see reference manual entry of `scale_image`.
- `HRegion operator >= (const HImage &image) const`
  `HRegion operator <= (const HImage &image) const`
  Selecting all pixel with gray values brighter than or equal to (or darker than or equal to, respectively) those of the input image, see reference manual entry of `dyn_threshold`.
- `HRegion operator >= (double thresh) const`
  `HRegion operator <= (double thresh) const`
  `HRegion operator == (double thresh) const`
  `HRegion operator != (double thresh) const`
  Selecting all pixel with gray values brighter than or equal to (or darker than or equal to, or equal to, or not equal to, respectively) a threshold, see reference manual entry of `threshold`.

Figure 3.5 gives an example of the use of the class `HImage`.

```
#include "HalconCpp.h"
#include "iostream.h"

main ()
{
  HImage   image("mreut");                    // Aerial image
  HWindow  w;                                 // Output window
  image.Display(w);                           // Display image
  // Returning the size of the image
  cout << "width  = " << image.Width();
  cout << "height = " << image.Height() << endl;
  // Interactive drawing of a region by using the mouse
  HRegion  mask = w.DrawRegion();
  // Reduce the domain of the image to the mask
  HImage   reduced = image & mask;
  w.ClearWindow();                            // Clear the window
  reduced.Display(w);                         // Display the reduced image
  // Applying the mean filter in the reduced image
  HImage   mean = reduced.MeanImage(61,61);
  mean.Display(w);
  HRegion  reg = bild >= (mean + 3);
  reg.Display(w);
}
```

Figure 3.5: Sample program for the use of the class `HImage` .

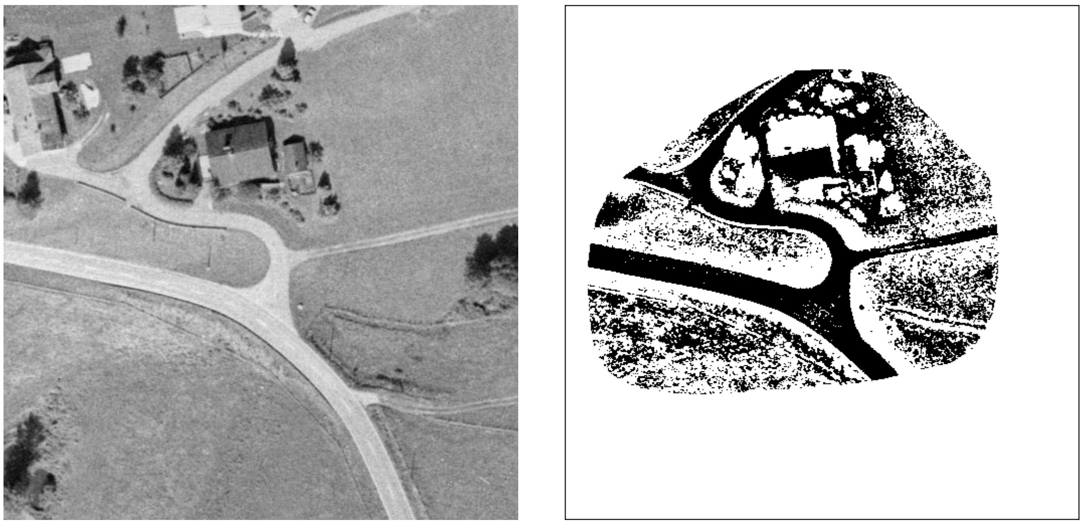

Figure 3.6: On the left side the input image (`mreut.tiff`), and on the right side the segmented regions in the selected image domain.

The example starts by reading a byte image from a file. The aim is to extract bright parts from the image. The used filter and the segmentation process itself is applied only in a pre-chosen part of the image in order to accelerate the runtime. This part is selected by drawing an arbitrary region with the mouse. This region `mask` serves as input for reducing the domain of the original image (operator `&`). The mean filter with a mask size of $61 \times 61$ is applied to the resulting region reduced. Bright pixels are selected by applying the operator >=. All pixels brighter than the filtered part of the image reduced $+3$ are selected. Figure 3.6 shows the result of the sample program in figure 3.5.

#### 3.1.2.2 Pixel Values

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

main ()
{
  HByteImage  in("mreut");          // Aerial image
  HWindow w;                        // Output window
  in.Display(w);                    // Displaying the image
  HByteImage out = in;              // Copying the image
  int  width  = out.Width();        // Width of the image
  int  height = out.Height();       // Height of the image
  long end    = width * height;     // Number of pixel of the image

  // 1. run: linear accessing
  for (long k = 0; k < end; k++) {
    int pix = in.GetPixVal(k);      // Reading the pixel
    out.SetPixVal(k,255-pix);       // Setting the pixel
  }
  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();

  // 2. run: accessing the image via the coordiantes (x,y)
  for (int y=0; y<height; y++) {
    for (int x=0; x<width; x++) {
      int pix = in.GetPixVal(x,y); // Reading the pixel
      out.SetPixVal(x,y,255-pix);  // Setting the pixel
    }
  }
  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();
}
```

Figure 3.7: Sample program for the use of the class HPixVal.

The class HPixVal is used for accessing the pixel values of the class HImage. Gray values can be set and returned independent of their types:

- HPixVal(void)
  Default constructor.
- HPixVal(const HComplex &Val)
  Constructing a pixel value from a complex number.

- `HPixVal(int Val)`
  Constructing a pixel value from an integer (`int`).
- `HPixVal(long Val)`
  Constructing a pixel value from a long (`long`).
- `HPixVal(HByte Val)`
  Constructing a pixel value from a byte (`byte`).
- `HPixVal(double Val)`
  Constructing a pixel value from a double (`double`).
- `HPixVal(const HPixVal &Val)`
  Copy constructor.
- `HPixVal &operator = (const HPixVal &grey)`
  Assignment operator.
- `operator HByte(void) const`
  Converting a pixel value to byte $(0 \dots 255)$.
- `operator int(void) const`
  Converting a pixel value to `int`.
- `operator long(void) const`
  Converting a pixel value to `long`.
- `operator double(void) const`
  Converting a pixel value to `double`.
- `operator HComplex(void) const`
  Converting a pixel value to `Complex`.

The handling of the class `HPixVal` is explained by an example in figure 3.7 which inverts an input image. The input image is a byte image. First, a copy is generated and the image size is determined. In the first run the pixels are accessed linearly. In the second run the pixel are accessed via the $(x, y)$-ccordinates.

#### 3.1.2.3 Image Arrays

The same way which was used to define arrays of regions is used to obtain arrays of images. The class is named `HImageArray` and contains the following member functions (in addition to the operators):

- `HImageArray(void)`
  Default constructor: empty array, no element.
- `HImageArray(const HImage &reg)`
  Constructing an image array from a single image.
- `HImageArray(const HImageArray &arr)`
  Copy constructor.
- `~HImageArray(void)`
  Destructor.

- `HImageArray &operator = (const HImageArray &arr)`
  Assignment operator.
- `long Num(void) const`
  Returning the number of elements in the array.
- `HImage const &operator [] (long index) const`
  Reading the element $i$ of the array. The index is in the range $0 \ldots \texttt{Num()} - 1$.
- `HImage &operator [] (long index)`
  Assigning an image to the element $i$ of the array. The index `index` can be $\geq$ `Num()`.
- `HImageArray operator () (long min, long max)`
  Selecting a subset between the lower `min` and upper `max` index.
- `HImageArray &Append(const HImage &image)`
  Appending another image to the image array.
- `HImageArray &Append(const HImageArray &images)`
  Appending another image array to the image array.

#### 3.1.2.4 Byte Images

For each pixel type, there exists a corresponding image class derived from `HImage`, e.g., `HByteImage` for the pixel type `byte` (standard 8 bit pixels) or `HInt2Image` for the pixel type `int2` (unsigned 16 bit pixels). The most important derived class is naturally `HByteImage`, as this pixel type still covers the majority of all applications in the field of image processing. The advantage of the class `HByteImage` in comparison to the class `HImage` is the simplified access to the pixel values. This is because the class `HPixVal` is not necessary. Besides the member functions of `HImage`, the class `HByteImage` contains the following extensions:

- `HByteImage(void)`
  Default constructor.
- `HByteImage(const char *file)`
  Constructing a byte image by reading a file.
- `HByteImage(int width, int height)`
  Constructing an empty byte image of a given size.
- `HByteImage(HByte *ptr, int width, int height)`
  Constructing a byte image by copying memory.
- `HByteImage(const HByteImage &image)`
  Copy constructor.
- `virtual ~HByteImage(void)`
  Destructor.
- `HByte &operator[] (long k)`
  Setting a pixel value by linear accessing.
- `HByte operator[] (long k) const`
  Reading a pixel value by linear accessing.

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

main ()
{
  HByteImage  in("mreut");         // Aerial image
  HWindow w;                       // Output window
  in.Display(w);                   // Displaying the image
  HImage out = in;                 // Copying the image
  int  width  = out.Width();       // Width of the image
  int  height = out.Height();      // Height of the image
  long end    = width * height;    // Number of pixel of the image

  // 1. run: linear accessing
  for (long k = 0; k < end; k++)
    out[k] = 255 - in[k];          // Reading and setting the pixel

  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();

  // // 2. run: accessing the image via the coordinates (x,y)
  for (int y=0; y<height; y++)
    for (int x=0; x<width; x++)
      out(x,y) = 255 - out(x,y);   // Reading and setting the pixel

  // Displaying the transformation
  cout << "Transformed !" << endl;  out.Display(w);   w.Click();
  cout << "Original !"    << endl;  in.Display(w);    w.Click();
}
```

Figure 3.8: Sample program for accessing a pixel value using the class `HByteImage`.

- `HByte &operator() (long k)`
  Setting a pixel value by linear accessing.
- `HByte operator() (long k) const`
  Reading a pixel value by linear accessing.
- `HByte &operator()(int x, int y)`
  Setting a pixel value by accessing it via $(x, y)$ coordinates.
- `HByte operator()(int x, int y) const`
  Reading a pixel value by accessing it via $(x, y)$ coordinates.
- `HByteImage operator & (int i)`
  Applying the logical “and”-operation on each pixel with `i`.
- `HByteImage operator << (int i)`
  Applying a left-shift on each pixel with `i`.

- `HByteImage operator >> (int i)`
  Applying a right-shift on each pixel with `i`.
- `HByteImage operator ~ (void)`
  Complement of each pixel.
- `HByteImage operator & (HByteImage &ima)`
  Pixel by pixel logical “and”-operation of two images.
- `HByteImage operator | (HByteImage &ima)`
  Pixel by pixel logical “or”-operation of two images.
- `HByteImage operator ^ (HByteImage &ima)`
  Pixel by pixel logical “xor”-operation of two images.

The advantage of the class `HByteImage` can be seen when accessing each pixel, see figure 3.8. The class `HPixVal` is not necessary in this example. Furthermore, the member functions `GetPixVal` and `SetPixVal` are not used. `HByteImage` allows to access pixel values in a notation like in the programming language C. The result of the example in figure 3.8 is basically the same as in the example in figure 3.7. The program in figure 3.8 is shorter, easy to read, and has a better runtime performance.

### 3.1.3 XLD Objects

*XLD* is the abbreviation for e**X**tended **L**ine **D**escription. This is a data structure used for describing areas (e.g., arbitrarily sized regions or polygons) or any closed or open contour, i.e., also lines. In contrast to regions, which represent all areas at pixel precision, XLD objects provide subpixel precision. There are two basic XLD structures: contours and polygons.

Similarly to images, HALCON/C++ provides both a base class `HXLD` and a set of specialized classes derived from `HXLD`, e.g., `HXLDCont` for contours or `HXLDPoly` for polygons. For all classes there exists a corresponding container class, e.g., `HXLDArray`.

In contrast to the classes described in the previous sections, the XLD classes provide only member functions corresponding to HALCON operators (see also section 2.1.2).

### 3.1.4 Low-Level Iconic Objects

As could be seen in the examples in chapter 2, when calling operators in the procedural approach, the class `Hobject` is used for all iconic parameters, be it an image, a region, or even an image array. In fact, the class `Hobject` is HALCON’s basic class for accessing the internal data management, i.e., it handles the keys of the database. Furthermore, `Hobject` serves as the basis for the class `HObject` and the derived classes, e.g., `HImage`.

The class `Hobject` has the following member functions:

- `Hobject(void)`
  Default constructor.
- `Hobject(const Hobject &obj)`
  Copy constructor.

- `virtual ~Hobject(void)`
  Destructor.
- `Hobject &operator = (const Hobject &obj)`
  Assignment operator.
- `void Reset(void)`
  Freeing the memory and resetting the corresponding database key.

As noted above, an instance of `Hobject` can also contain a tuple (array) of iconic objects. Unfortunately, `Hobject` provides no special member functions to add objects or select them; instead, you must use the operators `gen_empty_obj`, `concat_obj`, `select_obj`, and `count_obj` as described in section 2.1.3.

## 3.2 Control Parameters

HALCON/C++ can handle different types of alphanumerical control parameters for HALCON operators:

- discrete numbers (`long`),
- floating point numbers (`double`), and
- strings (`char *`).

A special form of control parameters are the so-called *handles*, which provide access to more complex data structures like windows, frame grabber connections, or models for shape-based matching. Internally, handles are almost always represented by discrete numbers (`long`); a notable exception is the handle to a bar code descriptor, which is an `HTuple`, as can be seen in figure 2.2. For handles there exist corresponding classes, which are described in section 3.2.3.

With the class `HTuple`, HALCON/C++ provides a container class for control parameters. What's more, `HTuple` is polymorphic, i.e., it can also contain arrays of control parameters of mixed type. To realize this, the auxiliary class `HCtrlVal` is introduced, which is described in the next section.

### 3.2.1 The Basic Class for Control Parameters

The class `HCtrlVal` serves as the basis for the class `HTuple` and is normally hidden from the user because it is only used temporarily for type conversion. The main point is that it can contain the three elementary types of control parameters, i.e., discrete numbers (`long`), floating point numbers (`double`), and strings (`char *`). `HCtrlVal` provides the following member functions:

- `HCtrlVal(void)`
  Default constructor.
- `HCtrlVal(long l)`
  Constructing a value from `long`.
- `HCtrlVal(int l)`
  Constructing a value from `int`.

- `HCtrlVal(double d)`
  Constructing a value from `double`.
- `HCtrlVal(const char *s)`
  Constructing a value from `char *`.
- `HCtrlVal(const HCtrlVal &v)`
  Copy constructor.
- `~HCtrlVal(void)`
  Destructor.
- `HCtrlVal& operator = (const HCtrlVal &v)`
  Assignment operator.
- `int ValType() const`
  Type of a value.
- `operator int(void) const`
  Conversion to `int`.
- `operator long(void) const`
  Conversion to `long`.
- `operator double(void) const`
  Conversion to `double`.
- `operator const char*(void) const`
  Conversion to `char *`.
- `double D() const`
  Accessing a value and conversion to `double`.
- `long L() const`
  Accessing a value and conversion to `long`.
- `int I() const`
  Accessing a value and conversion to `int`.
- `const char *S() const`
  Accessing a value and conversion to `char *`.
- `HCtrlVal operator + (const HCtrlVal &val) const`
  Adding two values.
- `HCtrlVal operator - (const HCtrlVal &val) const`
  Subtracting two values.
- `HCtrlVal operator * (const HCtrlVal &val) const`
  Multiplying two values.
- `HCtrlVal operator / (const HCtrlVal &val) const`
  Division of two values.

### 3.2.2 Tuples

The class `HTuple` is built upon the class `HCtrlVal`; it implements an array of dynamic length for instances of the class `HCtrlVal`. The default constructor constructs an empty array (`Num() == 0`). This array can dynamically be expanded via assignments. The memory management, i.e., reallocation, freeing, is also managed by the class. The index for accessing the array is in the range between $0$ and `Num()` $- 1$.

The following member functions reflect only a small portion of the total. For further information please refer to the file `HTuple.h` in `%HALCONROOT%\include\cpp`.

- `HTuple(void)`
  Default constructor. Constructs an empty tuple.
- `HTuple(long l)`
  Constructing an array of length $1$ from a discrete number `long` at index position $0$.
- `HTuple(int l)`
  Constructing an array of length $1$ from a discrete number converted to the internal type `long` at index position $0$.
- `HTuple(HCoord c)`
  Constructing an array of length $1$ from a coordinate at index position $0$.
- `HTuple(double d)`
  Constructing an array of length $1$ from a floating number `double` at index position $0$.
- `HTuple(const char *s)`
  Constructing an array of length $1$ from a string `char*` at index position $0$.
- `HTuple(const HTuple &t)`
  Copying a tuple.
- `HTuple(int length, const HTuple \&value)`
  Constructing an array of the specified length with a constant value, similar to the operator `tuple_gen_const`.
- `~HTuple()`
  Destructor.
- `HTuple &operator = (const HTuple& in)`
  Assignment operator.
- `HTuple Sum(void) const`
  Adding all elements in case they are numbers, similar to the operator `tuple_sum`.
- `HCtrlVal &operator [] (int i)`
  Setting the $i-$th element.
- `HCtrlVal operator [] (int i) const`
  Reading the $i-$th element.
- `HTuple operator + (const HTuple &val) const`
  Adding two tuples element by element, similar to the operator `tuple_add`. The arrays have to be of the same size.

- `HTuple operator + (double &val) const`
  `HTuple operator + (int &val) const`
  Adding a number to each element of the tuple, similar to the operator `tuple_add`.
- `HTuple operator - (const HTuple &val) const`
  Subtracting two tuples element by element, similar to the operator `tuple_sub`. The arrays have to be of the same size.
- `HTuple operator - (double &val) const`
  `HTuple operator - (int &val) const`
  Subtracting a number from each element of the tuple, similar to the operator `tuple_sub`.
- `HTuple operator * (const HTuple &val) const`
  Multiplying two tuples element by element, similar to the operator `tuple_mult`. The arrays have to be of the same size.
- `HTuple operator * (double &val) const`
  `HTuple operator * (int &val) const`
  Multiplying a number with each element of the tuple, similar to the operator `tuple_mult`.
- `HTuple operator / (const HTuple &val) const`
  Division of two tuples element by element, similar to the operator `tuple_div`. The arrays have to be of the same size.
- `HTuple operator / (double &val) const`
  `HTuple operator / (int &val) const`
  Division of each element of the tuple by a number, similar to the operator `tuple_div`.
- `HTuple Concat(const HTuple &t) const`
  Concatenating two tuples, similar to the operator `tuple_concat`.
- `extern ostream& operator<<(ostream &s, const HTuple &t)`
  Output of a tuple.
- `extern istream& operator>>(istream &s, HTuple &t)`
  Input of a tuple.

Figure 3.9 shows a short sample how to use tuples, i.e., the class `HTuple`: The default constructor generates an empty tuple. By assigning values to the tuple it is automatically expanded, and the data types of the values are also stored. For accessing the tuple the normal array notation can be used. If the data type of a value is not known in advance, an explicit type conversion has to be performed, see figure 3.9.

### 3.2.3 Classes Encapsulating Handles

The perhaps most prominent handle class is `HWindow`, which is described in section 3.2.3.1. Starting with version 6.1, HALCON/C++ also provides classes for handles to files or functionality like frame grabber access, measuring, or shape-based matching. See section 3.2.3.2 for an overview.

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

main ()
{
  HTuple  t;
  cout << t.Num() << '\n';             // The length of the tuple is 0
  t[0] = 0.815;                        // Assigning values to the tuple
  t[1] = 42;
  t[2] = "HAL";
  cout << t.Num() << '\n';             // The length of the tuple is 3
  cout << "HTuple = " << t << '\n';    // Using the << operator
  double d = t[0];                     // Accessing the tuple, if the
  long   l = t[1];                     // the types of the elements
  const char  *s = t[2];               // are known
  // Accessing the tuple, if the types of the elements are known
  printf("Values: %g %ld %s\n",t[0].D(),t[1].L(),t[2].S());
}
```

Figure 3.9: Sample for the use of the class `HTuple` .

#### 3.2.3.1 Windows

The class `HWindow` provides the management of HALCON windows in a very convenient way. The properties of HALCON windows can be easily changed, images, regions, and polygons can be displayed, etc. Besides those operators that can be called via `HWindow` (see also section 2.1.2), `HWindow` provides the following member functions:

- `HWindow(int Row=0, int Column=0,`
  `int Width=-1, int Height=-1,`
  `int Father = 0, const char *Mode = "",`
  `const char *Host = "")`
  Default constructor. The constructed window is opened.

- `~HWindow(void)`
  Destructor. This closes the window.

- `void Click(void) const`
  Waiting for a mouse click in the window.

- `HDPoint2D GetMbutton(int *button) const`
  `HDPoint2D GetMbutton(void) const`
  Waiting for a mouse click in the window. It returns the current mouse position in the window and the number of the button that was pressed, see the reference manual entry of `get_mbutton`.

- `HDPoint2D GetMposition(int *button) const`
  `HDPoint2D GetMposition(void) const`
  Returning the mouse position and the pressed button without waiting for a mouse click, see the reference manual entry of `get_mposition`.

- `HCircle DrawCircle(void) const`
  Waiting for the user to draw a circle in the window, see the reference manual entry of `draw_circle`.
- `HEllipse DrawEllipse(void) const`
  Waiting for the user to draw an ellipse in the window, see the reference manual entry of `draw_ellipse`.
- `HRectangle1 DrawRectangle1(void) const`
  Waiting for the user to draw a rectangle parallel to the coordinate axis in the window, see the reference manual entry of `draw_rectangle1`.
- `HRectangle2 DrawRectangle2(void) const`
  Waiting for the user to draw a rectangle with an arbitrary orientation and size in the window, see the reference manual entry of `draw_rectangle2`.

```
#include "HalconCpp.h"

main ()
{
  HImage  image("control_unit");      // Reading an image from a file
  HWindow w;                          // Opening an appropiate window
  image.Display(w);                   // Display the image
  w.SetLut("change2");                // Set a lookup table
  w.Click();                          // Waiting for a mouse click
  w.SetLut("default");                // Set the default lookup table
  w.SetPart(100,100,200,200);         // Set a part of the window
  image.Display(w);
  w.Click();
  // Adapting the part to the image again
  w.SetPart(0,0,bild.Height()-1,bild.Width()-1);
  image.Display(w);
  HRegionArray regs = image.Regiongrowing(1,1,4,100);
  w.SetDraw("margin");
  w.SetColored(6);
  regs.Display(w);
  w.Click();
  image.Display(w);
  w.SetShape("rectangle1");
  regs.Display(w);
}
```

Figure 3.10: Sample program for the use of the class `HWindow` .

Figure 3.10 shows the typical use of some member functions of the class `HWindow` and the different possibilities of displaying images and regions. The window is opened after reading the image from a file. This means, the window is scaled to the size of the image. The lookup table is changed afterwards, and the program waits for a mouse click in the window. A part of the image is zoomed now, and the program waits again for a mouse click in the window. By applying a region growing algorithm from the HALCON library (`Regiongrowing`) regions are generated and displayed in the window. Only the margin of the regions is displayed. It is displayed in 6 different colors in the window. The example ends with another way of displaying the shape of regions. The smallest rectangle parallel to the coordinate axes surrounding each region is displayed.

#### 3.2.3.2 Other Handle Classes

Starting with version 6.1, HALCON/C++ provides the following handle classes:

- `HBarCode1d`
  reading 1-dimensional bar codes, e.g., using the operator `Find1dBarCode`.
- `HBarCode2d`
  reading matrix bar code, e.g., using the operator `Find2dBarCode`.
- `HBgEsti`
  estimating the background of a scene, e.g., using the operator `RunBgEsti`.
- `HClassBox`
  classifying data, e.g., using the operator `TestSampsetBox`.
- `HFile`
  accessing disk files, e.g., using the operator `OpenFile`.
- `HFramegrabber`
  accessing frame grabbers, e.g., using the operator `GrabImage`.
- `HFunction1d`
  working with 1-dimensional functions, e.g., using the operator `CreateFunct1dPairs`.
- `HGnuplot`
  using gnuplot, e.g., using the operator `GnuplotPlotFunct1d`.
- `HMeasure`
  measuring distances along lines or arcs, e.g., using the operator `MeasurePairs`.
- `HOCR`
  optical character recognition, e.g., using the operator `DoOcrMulti`.
- `HOCV`
  optical character verification, e.g., using the operator `DoOcvSimple`.
- `HSerial`
  accessing a serial interface, e.g., using the operator `ReadSerial`.
- `HShapeModel`
  shape-based matching, e.g., using the operator `FindShapeModel`.
- `HSocket`
  accessing socket connections, e.g., using the operator `SendImage`.
- `HTemplate`
  gray value template matching, e.g., using the operator `BestMatchMg`.

- `HVariationModel`
  image comparison via a variation model, e.g., using the operator `CompareVariationModel`.

Besides the default constructor, the classes typically provide additional constructors based on suitable operators as described in section 2.1.2.1; e.g., the class `HBarCode1d` provides a constructor based on the operator `gen_1d_bar_code_descr`.

All handle classes listed above provide the methods `SetHandle()` and `GetHandle()`, which allow to access the underlying handle; furthermore, the classes provide an operator that casts an instance of the class into the corresponding handle. These methods are typically used when combining procedural and object-oriented code; for examples please refer to section 2.4.

## 3.3 Auxiliary Classes

In section 3.1.1, you already got a glimpse of additional classes provided by HALCON/C++: Instances of `HRegion` can be constructed from classes like `HDPoint2D` or `HRectangle1`. Currently, these classes are not documented in any of the manuals. We recommend to have a look at the header files in the directory `include/cpp`.

Please note that the header files in `include/cpp` include other classes, which do not appear in this manual. These classes are used by MVTec internally for testing purposes; they should not be used an application.

# Chapter 4

# Creating Applications With HALCON/C++

The HALCON distribution contains examples for building an application with HALCON/C++. Here is an overview of the relevant directories files (relative to `%HALCONROOT%`, Windows notation of paths):

`include\cpp\HalconCpp.h`:
: include file; contains all user-relevant definitions of the HALCON system and the declarations necessary for the C++ interface.

`bin\i586-nt4\halcon.lib,halcon.dll`:
: The HALCON library (Windows NT/2000/XP).

`bin\i586-nt4\halconcpp.lib,halconcpp.dll`:
: The HALCON/C++ library (Windows NT/2000/XP).

`bin\i586-nt4\parhalcon.lib,parhalcon.dll,parhalconcpp.lib,parhalconcpp.dll`:
: The corresponding libraries of Parallel HALCON (Windows NT/2000/XP).

`lib\%ARCHITECTURE%\libhalcon.so`:
: The HALCON library (UNIX).

`lib\%ARCHITECTURE%\libhalconcpp.so`:
: The HALCON/C++ library (UNIX).

`lib\%ARCHITECTURE%\libparhalcon.so,libparhalconcpp.so`:
: The corresponding libraries of Parallel HALCON (UNIX).

`include\cpp\HProto.h`:
: External function declarations.

`examples\cpp\makefile, makefile.nt`:
: Example makefiles which can be used to compile the example programs (UNIX and Windows NT/2000/XP, respectively).

`examples\cpp\make.%ARCHITECTURE%, macros.mak, rules.mak`:
: Auxiliary makefiles included by the makefiles listed above.

`examples\cpp\source\`
Directory containing the source files of the example programs.

`examples\cpp\bin\%ARCHITECTURE%\`
Destination of the example programs when compiled and linked using the makefiles.

`examples\cpp\i586-nt4\i586-nt4.dsw:`
Visual Studio workspace containing projects for all examples; the projects themselves are placed in subdirectories (Windows NT/2000/XP only).

`images\:`
Images used by the example programs.

`help\english.*:`
Files necessary for online information.

`doc\pdf\:`
Various manuals (in subdirectories).

There are several example programs in the HALCON/C++ distribution. To experiment with these examples we recommend to create a private copy in your working directory.

| | |
|---|---|
| `example1.cpp` | reads an image and demonstrates several graphics operators. |
| `example2.cpp` | demonstrates the direct pixel access. |
| `example3.cpp` | is an example for the usage of pixel iterators. |
| `example4.cpp` | demonstrates the edge detection with a sobel filter. |
| `example5.cpp` | solves a more complicated problem. |
| `example6.cpp` | is a very simple test program. |
| `example7.cpp` | demonstrates the generic pixel access. |
| `example8.cpp` | is an example for the usage of the tuple mode. |
| `example9.cpp` | introduces the XLD structure. |
| `example10.cpp` | demonstrates the usage of several contour structures. |
| `example11.cpp` | is another simple example for the usage uf tuples. |
| example_errorhandling.cpp | demonstrates the C++ exception handling (see section 2.2.1). |
| bottle2.cpp | recognizes on numbers on a beer bottle (OCR). |
| ean13.cpp | reads an EAN13 bar code. |
| ecc200.cpp | reads a Data Matrix (ECC200) code. |
| engraved2.cpp | recognizes engraved characters (OCR). |
| fuzzy_measure_pin.cpp | measures distances between pins using the fuzzy measure tool. |
| multi_chars.cpp | performs optical character verification (OCV). |

| | |
|---|---|
| pen.cpp | uses shape-based matching and the variation model for print quality inspection. |
| xing.cpp | monitors traffic using background estimation (by Kalman filtering). |

Additional examples for using HALCON/C++ can be found in the subdirectories `examples\mfc` and `examples\motif`.

In the following, we briefly describe the relevant environment variables; see the manual Getting Started with HALCON for more information, especially about how to set these variables. Note, that under Windows NT/2000/XP, all necessary variables are automatically set during the installation.

While a HALCON program is running, it accesses several files internally. To tell HALCON where to look for these files, the environment variable `HALCONROOT` has to be set. `HALCONROOT` points to the HALCON home directory. `HALCONROOT` is also used in the sample makefile.

The variable `ARCHITECTURE` describes the platform HALCON is used on. The following table gives an overview of the currently supported platforms and the corresponding values of `ARCHITECTURE`.

| ARCHITECTURE | Operating System (Platform) | Compiler |
|---|---|---|
| `i586-nt4` | Windows NT 4.0, Windows 2000, Windows XP on Intel Pentium or compatible | Visual Studio |
| `i586-linux2.2` `i586-linux2.2-gcc32` | Linux 2.2/2.4 on Intel Pentium (or compatible) | `gcc 2.95` `gcc 3.2/3.3` |
| `sparc-sun-solaris7` | Solaris 7 on Sparc Workstations | `CC` |
| `mips-sgi-irix6.5` | IRIX 6.5 on SGI Workstations (Mips processors) | `CC` |
| `alpha-compaq-osf5.1` | Tru64 UNIX 5.1 [1] on Alpha processors | `cxx` |

If user-defined packages are used, the environment variable `HALCONEXTENSIONS` has to be set. HALCON will look for possible extensions and their corresponding help files in the directories given in `HALCONEXTENSIONS`.

Two things are important in connection with the example programs: The default directory for the HALCON operator `read_image` to look for images is `%HALCONROOT%\images`. If the images reside in different directories, the appropriate path must be set in `read_image` or the default image directory must be changed, using `set_system("image_dir","...")`. This is also possible with the environment variable `HALCONIMAGES`. It has to be set before starting the program.

The second remark concerns the output terminal under UNIX. In the example programs, no host name is passed to `open_window`. Therefore, the window is opened on the machine that is specified in the environment variable `DISPLAY`. If output on a different terminal is desired, this can be done either directly in `::open_window(...,"hostname",...)` or by specifying a host name in `DISPLAY`.

[1] formerly called DIGITAL UNIX

In order to link and run applications under UNIX, you have to include the HALCON library path `$HALCONROOT/lib/$ARCHITECTURE` in the system variable `LD_LIBRARY_PATH`.

## 4.1 Creating Applications Under Windows NT/2000/XP

Your own C++ programs that use HALCON operators must include the file `HalconCpp.h`, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C++ interface. Do this by adding the command

```
#include "HalconCpp.h"
```

near the top of your C++ file. In order to create an application you must link the library `halconcpp.lib/.dll` to your program.

The example projects show the necessary Visual C++ settings. For the examples the project should be of the WIN 32 ConsoleApplication type. Please note that the Visual C++ compiler implicitly calls “Update all dependencies” if a new file is added to a project. Since HALCON runs under UNIX as well as under Windows NT/2000/XP, the include file `HalconCpp.h` includes several UNIX-specific headers as well if included under UNIX. Since they don’t exist under NT, and the Visual C++ compiler is dumb enough to ignore the operating-system-specific cases in the include files, you will get a number of warning messages about missing header files. These can safely be ignored.

Please assure that the stacksize is sufficient. Some sophisticated image processing problems require up to 6 MB stacksize, so make sure to set the settings of your compiler accordingly (See your compiler manual for additional information on this topic).

If you want to use Parallel HALCON , you have to link the libraries `parhalcon.lib/.dll` and `parhalconcpp.lib/.dll` instead of `halcon.lib/.dll` and `halconcpp.lib/.dll` in your project.

! Please note that **within an application you can use only one HALCON language interface**, be it directly or indirectly, e.g., by including a DLL that uses a second interface. Thus, you cannot use both the HALCON/C++ and the HALCON/COM interface in one and the same application.

## 4.2 Creating Applications Under UNIX

Your own C++ programs that use HALCON operators must include the file `HalconCpp.h`, which contains all user-relevant definitions of the HALCON system and the declarations necessary for the C++ interface. Do this by adding the command

```
#include "HalconCpp.h"
```

near the top of your C++ file. Using this syntax, the compiler looks for `HalconCpp.h` in the current directory only. Alternatively you can tell the compiler where to find the file, giving it the `-I<pathname>` command line flag to denote the include file directory.

To create an application, you have to link two libraries to your program: The library `libhalconcpp.so` contains the various components of the HALCON/C++ interface. `libhalcon.so` is the HALCON library.

Please take a look at the example makefiles for suitable settings. If you call `gmake` without further arguments, the example application `ean13` will be created. To create the other example applications (e.g., `example2`), call

```
gmake example2
```

You can use the example makefiles not only to compile and link the example programs but also your own programs (if placed in the subdirectory `source`). For example, to compile and link a source file called `myprogram.cpp` call

```
gmake myprogram
```

You can link the program to the Parallel HALCON libraries by adding `PAR=1` to the make command, for example

```
gmake myprogram PAR=1
```

Please note that **within an application you can use only one HALCON language interface**, be it directly or indirectly, e.g., by including a library that uses a second interface. Thus, you cannot use both the HALCON/C++ and the HALCON/C interface in one and the same application.

!

# Chapter 5

# Typical Image Processing Problems

This chapter shows the power the HALCON system offers to find solutions for image processing problems. Some typical problems are introduced together with sample solutions.

## 5.1 Thresholding an Image

Some of the most common sequences of HALCON operators may look like the following one:

```
HByteImage   Image("file_xyz");
HRegion      Threshold         = Image.Threshold(0,120);
HRegionArray ConnectedRegions = Threshold.Connection();
HRegionArray ResultingRegions =
               ConnectedRegions.SelectShape("area","and",10,100000);
```

This short program performs the following:

- All pixels are selected with gray values between the range 0 and 120.
- A connected component analysis is performed.
- Only regions with a size of at least 10 pixel are selected. This step can be considered as a step to remove some of the noise from the image.

## 5.2 Edge Detection

For the detection of edges the following sequence of HALCON/C++ operators can be applied:

```
HByteImage Image("file_xyz");
HByteImage Sobel = Image.SobelAmp("sum_abs",3);
HRegion    Max   = Sobel.Threshold(30,255);
HRegion    Edges = Max.Skeleton();
```

A brief explanation:

- Before applying the sobel operator it might be useful first to apply a low-pass filter to the image in order to suppress noise.

- Besides the sobel operator you can also use filters like `EdgesImage`, `PrewittAmp`, `RobinsonAmp`, `KirschAmp`, `Roberts`, `BandpassImage`, or `Laplace`.
- The threshold (in our case 30) must be selected appropiately depending on data.
- The resulting regions are thinned by a `Skeleton` operator. This leads to regions with a pixel width of 1.

## 5.3 Dynamic Threshold

Another way to detect edges is e.g. the following sequence:

```
HByteImage Image("file_xyz");
HByteImage Mean      = Image.MeanImage(11,11);
HRegion    Threshold = Image.DynThreshold(Mean,5,"light");
```

Again some remarks:

- The size of the filter mask (in our case $11 \times 11$) is correlated with the size of the objects which have to be found in the image. In fact, the sizes are proportional.
- The dynamic threshold selects the pixels with a positive gray value difference of more than 5 (brighter) than the local environment (mask $11 \times 11$).

## 5.4 Texture Transformation

Texture transformation is useful in order to obtain specific frequency bands in an image. Thus, a texture filter detects specific structures in an image. In the following case this structure depends on the chosen filter; 16 are available for the operator `TextureLaws`.

```
HByteImage Image("file_xyz");
HByteImage TT   = Image.TextureLaws(Image,"ee",2,5);
HByteImage Mean = TT.MeanImage(71,71);
HRegion    Reg  = Mean.Threshold(30,255);
```

- The mean filter `MeanImage` is applied with a large mask size in order to smooth the “frequency” image.
- You can also apply several texture transformations and combine the results by using the operators `AddImage` and `MultImage`.

## 5.5 Eliminating Small Objects

The following morphological operator eliminates small objects and smoothes the contours of regions.

```
          ...
::segmentation(Image,&Seg);
HCircle Circle(100,100,3.5);
HRegionArray Res = Seg.Opening(Circle);
```

- The term `::segmentation()` is an arbitrary segmentation operator that results in an array of regions (`Seg`).
- The size of the mask (in this case the radius is 3.5) determines the size of the resulting objects.
- You can choose an arbitrary mask shape.

## 5.6 Selecting Oriented Objects

Another application of morphological operators is the selection of objects having a certain orientation:

```
         ...
::segmentation(Image,&Seg);
HRectangle2 Rect(100,100,0.5,21,2);
HRegionArray Res = Seg.Opening(Rect);
```

- Again, `::segmentation()` leads to an array of regions (`Seg`).
- The width and height of the rectangle determine the minimum size of the resulting regions.
- The orientation of the rectangle determines the orientation of the regions.
- Lines with the same orientation as `Rect` are kept.

## 5.7 Smoothing Contours

The last example in this user's manual deals again with morphological operators. Often the margins of contours have to be smoothed for further processing, e.g. fitting lines to a contour. Or small holes inside a region have to be filled:

```
         ...
::segmentation(Image,&Seg);
HCircle Circle(100,100,3.5);
HRegionArray Res = Seg.Closing(Circle);
```

- Again, `::segmentation()` leads to an array of regions (`Seg`).
- For smoothing the contour a circle mask is recommended.
- The size of the mask determines how much the contour is smoothed.

# Index