Presentation PCL basics

Niveau 1

The programming language

Presentation PCL Basics

Version April 2013

Table of content

Introduction

Presentation exists of two programming languages SDL(scenario description language) and PCL(presentation control language). Presentation relies a lot on SDL. That is where this document comes in. SDL has some not usable timing features. There are two ways to program an experiment in Presentation:

- Mainly in SDL
- Mainly in PCL, with as little SDL as possible

We choose number 2, programming in PCL gives you maximum flexibility. The help file function in Presentation is not very supportive when you program PCL, but if you search a bit harder you will find that PCL is also well documented. So totally forget SDL programming and we start of with some basic programming skills in PCL. Concentrate on the constructions described, we will use this in the upcoming course. If you don't understand parts of the topics be sure to discuss it in the upcoming meeting, but keep on reading the rest of the document.

1. Basic PCL Types

PCL contains four basic variable types and many reference variable types. The four basic types are as follows:

- **int** : signed integer variables
- **double** : signed floating point variables
- **bool** : boolean variables
- **string** : character string variables

1.a Names and Declarations

Before you can use a variable, you must declare it. A variable declaration is a statement that has the following form:

type tVariableName [= expression]

with the part in brackets optional. This creates a variable named "tVariableName" of type "type". A variable name can be any combination of numbers, letters and underscores ('_') that begins with a letter. However, you can't use a reserved word, built-in function name or previously declared variable name.

```
#------- Example -------

int iValue;
double dValue;
int iCorrectInARow;
string sStimulusName;
```

String variables are initialized to be null strings. However "int", "double", and "bool" variables have undefined values after declaration. You may optionally assign a value to the variable when you declare it. The value can be any expression of the correct type.

```
#------- Example -------

int iValue = 5;
double dValue = 10.9;
double dNewVolume = dValue * 0.054;
string sMyName = "Bob";
```

Boolean literals are the reserved words ‘true’ and ‘false’. String literals must all be on one line (unlike in scenario files) but may contain the following escape characters: '\n' for a new line, '\t' for a tab, and '\"' for a double quotation mark. You can also specify the ascii code of a character by using '\x' followed by a two digit hexadecimal number or a unicode character by using '\u' followed by a four digit hexadecimal number.

```
#------- Example -------

string sIntro = "line 1: \"quote\"\nline 2";
bool bDoOver = false;
string sControl = "\x05\xa5\xb4\xb9";
string sUnicodeStr = "Chinese char: \u9F4B";
```

Floating point literals between 1 and -1 must have a zero before the decimal point and all floating point literals must have a decimal point. This includes zero, so that "0" is an integer literal while "0.0" is a double.

```
#------- Example -------

int iValue = 0;
double dValue = 0; # Error, 0 is not a double: dValue = 0.0;
dValue = .5; # Error, need a 0 before: dValue = 0.5;
iValue = 5e6; # power of 10, d = 5e6; # Error
d = 5.0e10;
```

You may also use character literals which have an integer value equal to their character codes. Character literals consist of a single character between single quotation marks. You may also use the escape characters '\n', '\t', '\".

```
#------- Example -------

system_keyboard.set_delimiter( ' ' ); # space-bar
```

1.b Conversions

PCL has static type checking with no implicit conversions between types. However, there are built-in functions that can be used to convert between basic types, for instance if you want to change a intereger to a double. The type names double as the function names in these cases:

```
#------- Example -------

int iValue = 5;
double dValue = double( i );
bool bBool = bool( i );
string sValueOfD = string( d );
```

We describe the exact behavior of the conversion functions in the Built-in Functions section.

1.c String methods

The string type has methods you can call for a string variable. A method is a function or subroutine that accesses the data for a specific variable. You call a method by typing the

variable name followed by a period (.) followed by the function name. The method may optionally require arguments.

```
#------- Example -------

# create a string
string sWords= "123456789";
# returns the number of characters in the string
int iSizeOfString = sWords.count();
```

2. Expressions

You can use operators (the standard mathematical signs) to manipulate your variables in Presentation. An expression is a statement that uses these operators to change the value of your variable.

```
#------- Example -------

int iValue = 8 * ( response_manager.hits() + 4 );
double dValue = -( 4.0 * random() - 4.0 );
my_sound.set_volume( dValue * 10.0 - 5.0 );
if (iValue == 1) || (jValue == 2) then
   mValue = 4
end;
```

If you want to combine variables, they need to be of the same type (e.g. **int** or **double**), otherwise you'll get an error. If you want to combine the values of different types of variables, you will have to converse them to the same type first.

```
#------- Example -------

double dValue = 0;        # Error, 0 is an integer
dValue = 0.0;             # Correct, type double with leading 0

dValue = 8 * random();    # Error, 8 is an integer and random() returns a double
dValue = 8.0 * random();  # Correct, 8.0 is a double

dValue = .5 * (4.0 + d2); # Error, .5 not a double literal
dValue = 0.5 * (4.0 + d2); # Correct, you must include the leading 0

int iValue = 5;           # start with type int
dValue = 10.0 * double( i ); # convert to double
```

Operators have the following precedence from highest to lowest:

```
! (not)
* (multiplication), / (division), %(mod)
+ (addition), - (subtraction)
== (equals), != (not equals), > (greater than), < (less than),
>= (greater than or equal to), <= (less than or equal to), && (and), || (or)
```

Operators with the same precedence are evaluated from left to right. Parentheses can be used to group subexpressions together and to alter the order of precedence in an expression. They also can increase clarity, facilitating reading and debugging of your experiment.

```
#------- Example -------

int iValue = 8 + 5 * 10;
iValue = (8 + 5) * 10;   # different precedence and different result

bool bBool = iValue == 5 && iAnotherValue == 4;
bBool = (iValue == 5) && (iAnotherValue == 4);   # the same result, increased readability
```

The following operators apply to the listed types:

```
int: +, -, *, /, <, >, <=, >=, == ,! =, %
double: +, -, *, /, <, >, <=, >=, ==, !=
bool: ==, !=, &&, ||, !
string: +, ==, !=
reference types: ==, !=
```

For **bool** values, using && (and) or || (or) results in both expressions being evaluated, regardless of the result of the evaluation of the first expression.

For string values, the '+' operator produces string concatenation.

```
#------- Example -------

string s1 = "Hello ";
string s2 = "there!";
```

```
string s3 = s1 + s2;   # "Hello there!"
```

3. Special Types

PCL has many other reference variable types, which are often dedicated to perform a specific function. You can view a complete list in the PCL Reference section of the Presentation Help. Only some of these types correspond to stimulus objects that can be defined in SDL. The remaining types give you access to other information and services provided by Presentation. For example, the **program_response_manager** type gives you access to information about responses that occur during the scenario.

The value of a reference variable always points to an object. For reference types related to SDL stimulus objects, the objects are created by your definitions in SDL. The remaining PCL objects are created in three general ways:

- Presentation automatically creates a reference variable referring to an object
- A reference variable can be obtained from another PCL object
- You can create your own objects using a special new operator

In the first case, Presentation automatically provides a variable you can use to access the services provided by an object. For example, the predefined variable **response_manager** of already points to an object. Just use this variable when you want to call one of the **program_response_manager** methods. (See **program_response_manager** PCL type.)

```
#------- Example -------

p_Picture1.present();
if (response_manager.hits() > 0) then
  p_Feedback1.present()
end;
```

If you look at the **program_response_manager** PCL type reference page, you will see the name of the predefined variable after "Predefined variable:". This will be true of all reference types with predefined variables. Since there is only one **program_response_manager** object allowed, there is no point to making more reference variables of this type. Thus, this is not allowed and is indicated on the reference page by "declarable: no - arrayable: no". The text "SDL defineable: no" means that objects of that type cannot be created by defining them in SDL.

Lastly, objects of some types can be created during a PCL program. To do this, use the keyword “new” followed by the name of the object type. This will create a new object of that type and return a reference to it. You will want to assign this to a variable of the appropriate type. Sometimes the new statement requires arguments. This will be indicated in the PCL type reference page for each type.

```
#------- Example -------

input_file InputFile = new input_file;
InputFile.open( "stim_sequence.txt" );
```

4. PCL Loops

The loop construct allows you to create PCL code that will execute a single section of code repeatedly. Every loop construct first contains a boolean expression. Each time Presentation executes the loop it evaluates the boolean expression. If the boolean expression evaluates to false, Presentation executes the code. If true, Presentation exits the loop.

```
loop
    statement_list
until
    boolean_expression
begin
    statement_list
end
```

A "statement_list" can contain any number of PCL statements separated by semicolons. The first statement list after the loop keyword typically sets a value for an initial condition or declares any variables only used inside the loop. Assume that we want to present 100 pictures to a subject, each of which is stored in an array. We might set a counter iValue to the value of 1 initially because we will start with the first array element.

The boolean_expression is what the loop uses to determine whether or not to run the statements between begin and end. For this experiment if we use the code ‘iValue > 50’ we will only pass through half of the array.

The statements that do the real work lie between the **begin** and **end** keywords. If you are using a counter in the loop, make sure you change the value in this statement list.

We use these statements to create the next example:

```
#------- Example -------

loop
        int iValue = 1;    # used for counter

until
        iValue > 100     # this is our boolean_expression
begin
    # present the correct picture in the array
        t_IntroText.set_caption( sWords[ iValue ] );
        ...
        p_Fixation.present();
        iValue = iValue + 1     # we increment the counter by one for each loop
end;
```

When PCL encounters a loop statement, it first executes the statement list between loop and until. In this case we set iValue to 1. It then evaluates the expression between until and begin. If the expression is false, the statement list between begin and end is executed. Inside the statement list we increment iValue by 1 each time it executes. When this is completed, the Boolean expression is evaluated again. Therefore the main body executes 100 times.

If you are using a counter or some other variable to determine when to exit the loop, make sure you update that variable in the body of the loop. If you forget to do this, the loop will run forever.

```
#------- Example -------

loop
        int iValue = 1
until
        iValue > sWords.count()
begin
        t_IntroText.set_caption( sWords[ iValue ] );
end;
# Oops! Runs sWords[1] forever
```

The break statement can be used to exit from a loop immediately, regardless of the loop conditional. When a break statement is encountered, execution resumes after the loop.

```
#------- Example -------

loop
        int iValue = 1
until
        iValue > sWords.count()
begin
        sWords[i].present();
        if (response_manager.last_response() == 2) then
            break;          # don't run remainder of trials
        end;
        iValue = iValue + 1
end;
```

The continue statement causes the remainder of the loop body to be skipped; however, it does not exit the loop. After a continue statement, the loop conditional is evaluated and if it is false, the loop body is executed again.

```
#------- Example -------

loop
        int iValue = 0
until
        iValue > sWords.count()
```

```
begin
        iValue = iValue + 1;
        sWords[i].present();
        if (response_manager.last_response() == 2) then
                continue;          # don't run trial2 and other stuff
        end;
        p_Picture2.present();
        # ...
end;
```

4.a Scope

All statements contained in a loop statement between loop and end form a scope. This means that variables declared in a statement inside the loop statement have no meaning outside the scope.

```
#------- Example -------

loop
        int iValue = 1
until
        iValue > iTest
begin
          ...
end;
iTest = iValue;   # Error, can't use iValue out here
```

5. PCL If Statements

The if-then construct allows you to build conditional statements. You use the if-then statement when you have boolean (true/false) choices within your experiment that delineate different conditions. For instance, you may want to play a high tone if the subject responded correctly to the trial and play a low tone if he responded incorrectly.

The if-then statement uses this template:

```
if boolean_expression then
    statement_list
[ elseif_part ]
[ else_part ]
end
```

Items in brackets are optional and are explained below. For each conditional block there must be exactly one [if boolean_expression then] and one end.

"boolean_expression" is any mathematical or logical expression that evaluates to either true or false (e.g. iValue < 1).

A "statement_list" is zero or more PCL statements separated by semi-colons.

The "else_part" of the statement is optional. If included it occurs exactly once. It must be the final statement of the conditional block and it represents the default action of the block if no other conditions evaluate to true. The "else_part" has the following form:

```
else_part: else statement_list
```

You may want to evaluate more than one condition in a conditional block. For instance, assume you create a reaction time experiment. You may want to execute different sets of instructions depending upon the value of a subject's reaction time for each trial. There is a condition for reaction times less than 100 ms (these values are too small for real reaction times and represent a guess). A second condition occurs if the reaction time is more than 4000 ms (these values are large and we assume that the subject missed the trial). Finally you may want to determine error rates for responses in the range of 100 ms to 1000 ms and compare them to error rates for values between 1001 ms to 3999 ms. You should implement the "elseif_part" to separate the final two conditions.

```
elseif_part: elseif boolean_expression then statement_list
```

This first example uses only the if-then construct.

```
#------- Example -------

if (iValue== 1) then
   iOutput = 2        # semi-colon not required
end;
```

In this example we add a default condition. If the if-then condition evaluates to false we assign 1.0 to the variable d

```
#------- Example -------

#   if the if-then statement is false
if (d > 5.0) then
   p_Fixation.present();  # semi-colon required
   d = 2.0
else
  p_Picture1.present();  # this is the default condition
   d = 1.0
end;
```

In the third example we need to evaluate three conditions before providing a default else condition.

```
#------- Example -------

if (iValue == 1) then
   p_Fixation.present;
   iOutput = 0
elseif (iValue == 2) then
    p_Picture1.present;
   iOutput = 1
elseif (iValue > 2) then
   iOutput = 2
else
   iOutput = 3
end;
```

This final example has multiple conditions but if both conditions are false there is no default action.

```
#------- Example -------

if (iInput != 1) then
   p_Picture1.set_part_x( 1, 2 )
elseif (iCondition > 0) then
   p_Picture1.set_part_x( 1, l )
```

```
end;
```

The sections between keywords then, elseif, else, and end are statement lists so the final statement in the list does not require a trailing semicolon although it is useful to add one in case you add code later. The keyword elseif is one word which may be confused with the two words "else if". If you incorrectly substitute "else if" for elseif, your if statement will be missing an end. (We provide an example below why this is incorrect).

```
#------- Example -------

if (values[iCounter] == m) then
      iOutput = iOutput + 1
else if (values[iCounter] == 2) then #Oops!
      iOutput = iOutput - 1
end;

# The following spacing addresses the "else if" problem
if (iCounter == 1) then
      iOutput = iOutput + 1
else
     if (iCounter == 2) then
           iOutput = iOutput – 1
     end;
# Missing "end" here! Should have used "elseif"
```

5.a Scope

Each of the statement lists contained in an if statement is a separate scope. The scope of a variable means that variables declared in one of those statement lists can only be used within that statement list and does not exist outside of it. You can use outside variables in those statement lists. Currently PCL does not allow you to override outside variables.

```
#------- Example -------

int iValue = 5;
if (iInput > 3) then
    int iOutput = 0;
    ...
    int iValue;    # Error, can't override iValue
else
    iOutput = 2;    # Error, iOutput doesn't exist in this scope
    ...
end;
iValue = 5;       # Error, iValue no longer exists
```

6. Arrays

PCL programs can use variables that are arrays of other variables. You can store the four basic variable types as well as some of the reference types in arrays. We say a variable type is "arrayable" if it can be stored in an array. You can find out which reference types are arrayable in the PCL Reference section and going to the page of the particular type you are interested in.

An array declaration has the following form:

```
array <arrayable_type> name[integer_valued_expression]
```

The expression between the brackets specifies the size of the array.

```
#------- Example -------

int iValue = 5;
array <int> aiMyIntegers[iValue];
array <picture> apMyPictures[iValue];
```

Arrays may also be multi-dimensional. For multi-dimensional arrays, additional ranges are placed between brackets:

```
#------- Example -------

array <int> aiMyIntegers[10][10][5];
```

You may assign all elements of an array with literals of that type in one statement if the elements are one of the four standard types. The elements in an array literal are listed between curly brackets, are separated by commas and must be literals. You can create a multi-dimensional array literal by nesting the curly brackets.

```
#------- Example -------

array <int>
aiValue[3] = { 1, 2, 3 };
aiIntegers = { 4, 5, 6 };

array <int>
aiMoreIntegers [2][3] = { { 1, 3, 3 }, { 4, 5, 6 } };

int iValue = 1;
aiIntegers = { iValue, iValue, iValue }; # Error! Elements of an array literal must be literals
```

Elements of an array variable are accessed by placing an integer index between square brackets after the array name. The integer index can be any expression with an integer value. The indices of array elements start with 1. Incomplete indexing of a multi-dimensional array will yield another array of lower dimension.

```
#------- Example -------

int i = aiMyInts[5];
picture p_NextPicture = apMyPictures[ 2 * i ];

int j = aiMyInts[0]; # Error! Array indices start with 1

array <int> aiMoreInts[3][10];

aiMoreInts[1] = aMyInts;
aiMoreInts [2] = aMyInts;
aiMoreInts [3] = aMyInts;

i = aiMoreInts[2][random( 1, 10 )];
```

When copying one array onto another the arrays must have the same type and dimensionality but they need not have the same size. If the sizes differ, the maximum number of possible assignments are performed. If you initialize an array in the declaration statement with an array of a smaller size, the remaining elements will not be zeroed.

7. Include Statements

You can use an include statement to read PCL code from a file in addition to the main PCL file or scenario file. The include statement has the following form.

```
include "filename"
```

This statement causes Presentation to insert the contents of the specified file at that point in the PCL program. If the filename you provide does not contain a complete path, the filename is relative to the directory of the file containing the include statement.

```
#------- Example -------

include "C:\\experiments\\common\\standard_subroutines.pcl";
include "utility.pcl"; # same directory as this file
```

Remember that you need two backslashes within a string to result in a single backslash.

8. Subroutines

8.a Introduction

Subroutines are useful in converting what are possibly extended, complicated lines of code for a complex experiment into logical subgroups that allow for a greater flexibility and easier maintenance. Properly implemented, subroutines provide a modular structure to PCL code.

You may define your own subroutines that can be called later in your PCL program. You may pass arguments and receive return values from subroutines. A subroutine is defined in a subroutine statement. A subroutine statement has the following form (optional parts inside []):

```
sub
   [ return_value ] name [ argument_list ]
begin
   subroutine_body
end
```

The method you choose for spacing between words is not important. If the return value exists it will be the name of a PCL type. The argument list is a list of type name and variable name pairs separated by commas. The subroutine body is an arbitrary number of PCL statements except for other subroutine definition statements. Subroutines are called using the name of the subroutine followed by parentheses () enclosing any arguments.

```
#------- Example -------

sub
   ShowThreePictures   # this subroutine takes no arguments
begin
   p_Pic1.present();
   p_Pic2.present();
   p_Pic3.present()
end;

sub
   Wait( int iDuration )  # this takes one argument of type int
begin
   loop
      int iEndTime = clock.time()  + iDuration
   until
      clock.time() > = iEndTime
   begin
            # we don't execute any code in here
   end
end;
```

8.b Scope

The subroutine body forms a local scope for those variables declared in the argument list. These variables are only defined within the subroutine body. Statements in the subroutine body may reference any global variables declared before the subroutine definition.

```
#------- Example -------

int iIndex = 0;

sub
   AdjustIndex( int j )     # j is local to this subroutine
begin
   if (mod( j, 2 ) == 0) then
      iIndex = iIndex + 1
   else
      iIndex = iIndex - 1
   end
end;

j = 5;  # error, j doesn't exist out here
int j = 3;  #ok, declare a j outside of subroutine's scope
AdjustIndex( j ); # call subroutine and change index
```

In the example Presentation knows that the j inside the subroutine is different from that outside the subroutine.