Writing z/OS CGIs in COBOL

Writing z/OS CGIs in COBOL - Course Objectives

On successful completion of this class, the student, with the aid of the appropriate reference materials, should be able to:

1. Code, compile, bind, debug, deploy, and maintain CGIs for the z/OS environment, written in COBOL

2. Handle GET and POST requests: analyze and take action, as appropriate
   * Parse and decode a QUERY_STRING value for GET
   * Gather in the stdin data for POST
     - Save a file as is or translated to EBCDIC on the mainframe, for POST

3. Produce responses that are dynamically created HTML pages or redirection to existing pages

4. Access environment variables

5. Access DB2 data (optional: depends if DB2 installed and lab set up done)

6. Access VSAM KSDS data by primary key or alternate index

7. Put out HTML encoded in UTF-16, to provide a truly international aspect to your website

8 Submit jobs to the batch from a CGI (optional; may not be appropriate in all environments).

Note: Generally speaking, the comments here about HTML also apply to XHTML; but our focus is on using HTML 5

Note: This course supports the HTTP server provided free with z/OS and the ported Apache server (see page 4).

Writing z/OS CGIs in COBOL - Topical Outline

Writing z/OS CGIs in COBOL - Topical Outline, p.2.

HTTP Servers on z/OS

❒ There are several HTTP servers available for z/OS, but these are the known free choices (identified by the value found in the environment variable SERVER_SOFTWARE):

- ♦ IBM HTTP Server/V5R3M0 - this server comes free automatically with z/OS and is based on early standards (still works fine, though)

- ♦ IBM Apache Server - since late 2008, IBM provides this Apache server already ported, along with some ported tools; this is free but must be separately ordered

❒ There is also WebSphere Application Server (WAS) which comes with the same Apache server - but WAS is not free

❒ There is also a free Tomcat server from Dovetailed Technologies, which is Java-centric (http://www.dovetail.com/products/tomcat.html)

❒ For simplicity, we assume you are using one of the free available servers, which we shall refer to as "the HTTP server" (for the first server in the list above) or "Apache" for the second

- ♦ Technically, of course, these are all HTTP servers, but we're looking for a shorthand to be both concise and accurate

- ♦ Finally, since the behavior of these servers is largely the same, you can take "the server" to be shorthand for "either the HTTP server or the Apache server"

Section Preview

❒ **General Program Structure and Techniques**

- ♦ **General Program Structure**
- ♦ **Redirect Using display**
- ♦ **Redirect Using printf**
- ♦ **Redirect Using bpx1wrt**
- ♦ **Watching for Errors**
- ♦ **Deploying Your CGI**
- ♦ **Setting Up for Labs (Machine Exercise)**

General Program Structure

❒ **The main work of a CGI is writing out HTML pages to** stdout**, which are then intercepted by the HTTP server and transmitted to the requesting client**

❒ **There are three basic choices for writing to** stdout **from COBOL**

- ♦ **<u>Use COBOL DSPLAY statements</u> - the most natural**
- ♦ **<u>Use the callable service bpx1wrt</u> - which uses a classic MVS, z/OS approach in its parameters**
- ♦ **<u>Call the C function printf()</u> - a viable alternative; not as natural in COBOL as using display, but it handles certain formating chores more easily than display**
    - ✗ Especially if you have numeric data to display, or need character string data to be constructed from strings whose length you don't know in advance

❒ **CGIs written in COBOL must be compiled RENT and NODYNAM (reentrant code and static calls)**

- ♦ **Also, you may need to have the C compiler licensed to invoke the C functions** printf() **and** getenv() **that are useful in our work**
- ♦ **Still, you can always call kernel services such as bpx1wrt**
    - ✗ Except there is no kernel callable service for working with environment variables, so you must use either the C functions of the new CEEENV LE service for that

Redirect Using Display

❒ **For all COBOL CGIs, we use the typical COBOL program structure:**

```
 process nodynam xref(short) rent
* Copyright (C) 2009 by Steven H. Comstock
 Identification division.
 Program-id.  TCBREDD.
*
* COBOL program designed to run as a CGI
* the program simply redirects the server
*    to a different file
*

 Environment division.
 Data division.
 Working-storage section.

 01  pic x(16) value 'Ver1 of TCBREDD'.

 01  loc.
     02  pic x(22)
          value
         'Location: ../~scomsto/'.
     02  pic x(20)
          value  'customer.html.ascii'.
     02  nl  pic x(1) value x'15'.

 procedure division.
      display loc
      goback.
```

Redirect Using Display, 2

Notes

- **The 'process' statement ensures we compile reentrant and that calls are static**
  - ✗ Using dynamic calls for CGIs requires the use of the bpx1lod callable service or using DLL linkages
    - ➢ Not discussed here, to focus on CGI content and structure
- **The item labeled "loc" contains the text needed in a redirect header**
  - ✗ The ../~scomsto says "back out of the current directory" (the two dots) "then go into your user id's web pages directory" (/~ followed by your z/OS UNIX ID)
    - ➢ You will need to replace this with a similar construct using your actual ID, as described for you in the first lab writeup
  - ✗ And customer.html.ascii is the name of the page to redirect to in this directory
    - ➢ Notice the full name is spread across two lines; since they compile contiguously, the resulting string is what we need
- **The x'15' at the end of loc represents an EBCDIC new line (NL) character**
  - ✗ Since there's no output following, the HTTP server understands there are only headers in this transmission
  - ✗ Might be a little cleaner to send an extra blank line by adding:

```
display nl
```

Redirect Using printf

❒ **Now, to accomplish the same task with calls to printf:**

```
 process nodynam xref(short) rent
* Copyright (C) 2009 by Steven H. Comstock        Ver1
 Identification division.
 Program-id.  TCBREDP.
*
* COBOL program designed to run as a CGI
* the program simply redirects the server
*    to a different file
* using printf callable service
*

 Environment division.
 Data division.
 Working-storage section.

 01   pic x(16) value 'Ver1 of TCBREDP'.

 01   loc.
      02   pic x(22)
             value
               'Location: ../~scomsto/'.
      02   pic x(20)
             value  'customer.html.ascii'.
      02   blank-line pic xx value x'1500'.

 procedure division.

       call 'printf' using loc
       call 'printf' using blank-line

       goback.
```

- **Note that "blank-line" is a null-terminated string, which is what the C** printf **function requires**

Redirect using BPX1WRT

❒ **Now let's look at the same function using calls to the z/OS UNIX kernel service** bpx1wrt

❒ **If we want to use BPX1WRT, the general program structure is pretty much the same, but we have these issues to address**

- **The bpx1wrt service requires seven parameters, passed in the classic MVS, z/OS style:**

  ✗ File descriptor number; use a fullword binary 1 for **stdout**

  ✗ Address of a pointer to the buffer containing the data to write

  ✗ Address to a pointer to a buffer ALET (Address space or data space where buffer is); specify zeros to indicate the current address space, which is what we want

  ✗ Bytes to write - fullword binary integer containing the length of the data you want to put out

  ✗ Return value from function: -1 indicates write failed; otherwise the actual number of bytes written

  ✗ Return code and reason code; each fullwords; not meaningful unless return value is -1

- **This service must not have trailing nulls in its parameters**

Redirect using BPX1WRT, 2

❒ **So we end up with this code:**

```
*process nodynam xref(short) rent
* Copyright (C) 2009 by Steven H. Comstock       Ver1
 Identification division.
 Program-id.  TCBREDB.
*
* COBOL program designed to run as a CGI
* the program simply redirects the server
*    to a different file, using bpx1wrt callable service
*

 Environment division.
 Data division.
 Working-storage section.

 01  pic x(16) value 'Ver1 of TCBREDB'.

 01  loc.
     02  pic x(22)
          value  'Location: ../~scomsto/'.
     02  pic x(20)
          value  'customer.html.ascii'.
     02  blank-line pic x(1) value x'15'.

 01  bpx1-stuff.
     02  stdout          pic s9(8) binary value 1.
     02  buffer-ptr      pointer.
     02  buffer-alet     pic s9(8) binary value 0.
     02  num-bytes       pic s9(8) binary value 0.
     02  return-co       pic s9(8) binary value 0.
     02  reason-co       pic s9(8) binary value 0.
     02  return-val      pic s9(8) binary value 0.
```

Redirect using BPX1WRT, 3

```
procedure division.

      set  buffer-ptr to address of loc
      move length of loc to num-bytes
      call 'bpx1wrt' using stdout,
                                buffer-ptr,
                                buffer-alet,
                                num-bytes,
                                return-val,
                                return-co,
                                reason-co

      set  buffer-ptr to address of blank-line
      move 1 to num-bytes
      call 'bpx1wrt' using stdout,
                                buffer-ptr,
                                buffer-alet,
                                num-bytes,
                                return-val,
                                return-co,
                                reason-co

      goback.
```

- **Notice that the two calls to bpx1wrt are the same**
    - **It might make good sense to put the call into a separate paragraph and perform that paragraph after initializing the two variable fields**
        - And we use that approach later

Redirect Notes

- For all of these redirect-ing CGIs, the redirect address can be a fully-specified URI, for example:

```
01   loc.
      02   pic x(40)
             value
            'Location: http://192.168.1.231/~scomsto/'.
      02   pic x(20)
             value   'customer.html.ascii'.
      02   nl   pic x(1) value x'15'.
```

  - ... or ...

```
01   loc.
      02   pic x(nn)
             value
            'Location: http:domain_name/~scomsto/'.
      02   pic x(20)
             value   'customer.html.ascii'.
      02   nl   pic x(1) value x'15'.
```

  - The key here, in both cases, is to ensure the length of the first field exactly matches the length of the string, so that the second field value follows immediately

- Because it is so much simpler, we will use 'display' as our preferred way to emit HMTL (with occasional use of bpx1wrt and printf)

  - DISPLAY is inherent in the language, so we will only use the alternative methods when they are needed for functionality that DISPLAY may not have (for example, *printf*'s ability to work with null terminated strings easily)

Watching for Errors

❐ **Debugging CGIs is generally quite awkward**

- ♦ **The environment is complex**
- ♦ **Often the HTTP server tries to continue on, even after a CGI has abended**

❐ **So one step you can take in your code is to watch for errors**

❐ **For example, both printf and bpx1wrt allow you to test for success**

- ♦ **Add a RETURNING clause in your calls to** printf **- if you get any negative number, you had a problem**
- ♦ **Check the return code (return-co) after a call to bpx1wrt - if it is -1, you had a problem**

❐ **But what will you do in these cases? Well, you can try several approaches**

- ♦ **Use DISPLAY to write a message**
- ♦ **Call the LE service CEEMOUT to write to** stderr
- ♦ **Call bpx1wrt but write to** stderr **(use a fullword 2)**
- ♦ **Call CEE3ABD or CEE3AB2 with helpful user return codes**
- ♦ **Call CEE3DMP to display data items**

❐ **If you have errors with services other than printf or bbx1wrt, at least you can use bpx1wrt or** printf **to write out HTML text to the client that gives some indication of the situation**

Watching for Errors, 2

❒ In our code samples and labs we will not do extensive error checking, in order to focus on functionality

- ♦ But in a number of places we will demonstrate error checking and handling, so you can see some of the ways of dealing with errors

❒ When trying to debug CGIs, it is often helpful to look at the HTML the CGI has emitted up to the point of the error

- ♦ Using your browser to look at a page put out by your CGI, right click on a blank spot of the page
- ♦ In most browsers a pop-up menu will include an option like "View page source"
- ♦ Selecting this will show you the HTML your CGI wrote out, perhaps giving you some clues where things went wrong

❒ There is a pretty good tool for examining HTTP traffic; it's called HTTPLook, it's shareware and you can download it from

- ♦ http://www.brothersoft.com/httplook-download-25677.html
- ♦ Caution: download then run the install program; you will see a dialog about installing the BrotherSoft Extreme with some check boxes; close the dialog; when it prompts you to continue or exit setup, choose exit; wait a while and then you will see the setup dialog for HTTPLook - now install this program

Deploying Your CGI for Testing

- ❐ **Once you have your CGI coded, you need to compile and bind and put the code in the correct place for it to be found by the HTTP server when it is called for**

- ❐ **So the steps are:**

  - ♦ **Compile and bind using JCL (we shall bind into a PDSE named <your_id>.TR.PDSE)**

    - ✗ Or, if you are working under the shell, use the **cob2** command to compile and bind into your CGI directory

  - ♦ **If working outside of the shell, you need to copy your executable load module into your CGI directory; use ISPF 6 like this:**

```
 ===> oput tr.pdse(tcbredp) '/u/scomsto/CGI/tcbredp'
```

    - ✗ Note that for the second operand, <u>case is important</u>; also you need to specify the name of the directory set up for your CGIs instead of the directory shown, of course

  - ♦ **Either way, your last step here is to ensure the CGI program has the right permission bits; if you are not in the shell already, issue the omvs command from ISPF 6 then issue these commands:**

```
          cd CGI
          chmod 755 tcbredp
```

    - ✗ Note that you only have to do this the first time you put each CGI into your directory; later, if you replace it, the permission bits are remembered

Computer Exercise: Setting Up For Labs

This machine exercise is designed to provide setup for all the remaining class exercises.

In order to work with CGIs, a lot of pieces have to be in place:

* You must have the IP address or system name of your host where the CGIs will run; this can be internal (your intranet, behind your firewall) or external (your internet presence, accessible by browsers from outside your organization):

  ____________________________ (system name or IP address)

* You must have a z/OS UNIX ID, part of what's called an OMVS segment as part of your security package; this includes a user id for logon (a character string that is usually lower case), and a UID (an integer) to identify you to the user database, a home directory (usually of the form **/u/***user_id*), and some other information: ___________ (your user id)

* You must have a TSO id also (which we assume to be your z/OS UNIX userid in upper case); normally the password for both ids is the same.

* You must know your installation's choice for the directory where web pages should be stored; often it is **public_html** under your home directory; that is: **/u/***user_id***/public_html**, but not always:

  ________________________ (web page directory)

* You need to know the name of the directory where your CGIs should reside; it is often called cgi-bin or **CGI** and is under your home directory; that is:
  /u/*user_id***/CGI** but it does not have to be so:

  _______________ (CGI directory)

* Finally, you need to know the mapping id that the server will use to direct CGI requests to your CGI directory; for example, in our shop, our configuration file has the entry:

```
Exec        /SCOMSTO/*            /u/scomsto/CGI/*
```

  which says requests for any file in SCOMSTO should resolve to files in /u/scomsto/CGI, my CGI directory; SCOMSTO is my CGI mapping id.

  ______________ (CGI mapping id)

Computer Exercise, p.2.

For this lab, you have two parts: 1) the set up work and then 2) a small lab that will build on the lecture and test the set up at the same time.

The set up

Run uc04strt, a supplied REXX exec that will prompt you for the high level qualifier (HLQ) you want to use for your data set names; the exec uses a default of your TSO id, and that is usually fine. Then the exec creates data sets and copies members you will need. Then there is still some work to do.

From ISPF option 6, on the command line enter:

```
===> ex '__________.train.library(uc04strt)' exec
```

A panel displays for you to specify the HLQ for your data sets, with your TSO id already filled in. Press <Enter> and you get a panel telling you setup has been successful. Press <Enter> again and you are back to the ISPF command panel

The allocated data sets:

<hlq>.TR.CNTL	for your JCL (and it also contains some archive files and other data as members)
<hlq>.TR.COBOL	for your source code
<hlq>.TR.PDSE	for program objects
or	
<hlq>.TR.LOAD	for load modules

Computer Exercise, p.3.

Next, get into OMVS, and cd to your html directory and issue these commands:

```
umask 000
pax -r -f "//tr.cntl(uc04html)"
```

this unwinds the testing HTML pages and some data.

While you are in this directory, create a sub-directory we will use in a later lab:

```
mkdir PDFs
```

Also while you are in this directory, you should oedit the file **CGI_Labs.html** as follows:

* change all occurrences of SCOMSTO to the mapping id for your CGI directory

* If you have access to a corporate logo image file, you can change the <img > tag to point to that logo.

Next, change to your CGI directory, and issue this command:

```
pax -r -f "//tr.cntl(cgis)"
```

this unwinds your style sheet (discussed later).

Computer Exercise, p.4.

The lab.

Exit OMVS and get into edit of your source PDS. There are three members there that do redirects:

```
TCBREDD - redirect using display
TCBREDB - redirect using bpx1wrt
TCBREDP - redirect using printf
```

Modify each of these so that "scomsto" is changed to your id.

> Tip: watch out for the need to change the size of various fields when you make these kinds of changes, throughout the course.

Now compile and bind each of these. To do this, edit your TR.CNTL library, member COBCGIA. This JCL compiles and binds programs into your TR.PDSE library (or TR.LOAD if PDSEs are not supported). The

```
//   SET  O=
```

line should have the name of the member to compile and bind. So compile and bind each of these three source programs.

Once you have clean compiles and binds, deploy the executables from your TR.PDSE or TR.LOAD library to your CGI directory. (see page 16 for hints)

Finally, test your work by pointing your browser on your workstation to your CGI_Labs.html page and runnng the COBOL programs listed in the first test option.

<u>Conventions used in this course</u>:

192.168.1.231	- internal IP address used by course author for development and testing; always replace with your system name or IP address
scomsto	- UNIX id used by the author; always replace with your UNIX id
public_html	- directory for user HTML pages; always replace with your HTML directory
~scomsto	- mapping id used to get to your HTML directory; replace with your mapping id
SCOMSTO	- mapping id used to get to your CGI directory
CGI	- actual directory for user CGIs to run from; always replace with your CGI directory mapping id
/s-css/*	- directory for style sheets referenced by CGIs; maps to /u/scomsto/CGI/* ; always replace with your CGI stylesheet mapping (more later)

Conventions used in this course, 2:

CGI program names used in all our language-specific CGI courses: **TC**xfffs where:

T	comes from The Trainer's Friend		
C	indicates this is a CGI		
x	indicates the programming language; one of:		
		A	- Assembler
		B	- COBOL
		C	- C
		P	- PL/I
		X	- REXX
fff	mnemonic for the function, *e.g.*: RED for REDIRECT		
s	indicate method used to write to stdout; one of:		
		B	- BPX1WRT
		P	- printf()
		D	- display (COBOL)
		K	- put skip (PL/I)
		S	- say (REXX)
		E	- echo (shell script)
		R	- print (Perl, Java, php)
		X	- EXECIO (REXX)

In a few cases, we may not follow this naming convention but it will usually help you keep straight which program is which.

Section Preview

❒ **Basic Processing**

- **Emitting Headers**
- **Emitting HTML**
- **Accessing environment variables**
- **Displaying environment variables**
- **Stylesheets and CGIs**
- **Writing out HTML pages (Machine Exercise)**

Emitting Headers

- **Every CGI must emit**
  - **One or more HTTP headers**
  - **A blank line**
  - **Some content**
    - ✗ Usually an HTML page
      - ➢ Perhaps also some log or trace information or error messages

- **We saw with the redirect example a single header (Location) and a blank line**
  - **If no content is supplied with a redirect header, the z/OS HTTP server supplies a little content to help the transmission protocol be maintained**

Emitting Headers, 2

- ❐ When you are not just doing a Location header, most typically you emit a Content-type header

  - ♦ Using a content type of text/html, you can add two NL characters to send the header line and corresponding blank line

  - ♦ If you are using printf, you need a trailing null, so define:

```
01   content-hdr.
      02   pic x(24)
             value   'Content-type: text/html'.
      02   pic xxx   value x'151500'.
```

  - ♦ And write to stdout with:

```
           call 'printf' using content-hdr
```

  - ♦ If you are using bpx1wrt, define content-hdr without the trailing null and write out with:

```
       set   buffer-ptr to address of content-hdr
       move length of content-hdr to num-bytes
       call 'bpx1wrt' using stdout, buffer-ptr,
                           buffer-alet, num-bytes,
                           return-val, return-co,
                               reason-co
```

  - ♦ If you are using display, just write out literals:

```
       display 'Content-Type: text/html'
       display ' '
```

Emitting HTML

❒ **Now you may have some work to do before you start writing out your HTML, but you will, at some point, want to put out these lines:**

```
<!DOCTYPE html>
<html>
<head>
<link rel=stylesheet href=/s-css/cgi-style1.css
  type=text/css >
```

- ♦ **Then a title element, then the end of your <head> section, then start your <body>**

- ♦ **After your detail lines (body), you will want to bring closure with </body> and </html> before ending your CGI**

❒ **Notice the link to a stylesheet**

- ♦ **This is optional, of course, and there are some issues regarding style sheets, CGIs, and the HTTP server - which we address later in this section**

- ♦ **But having the ability to work with a stylesheet is pretty essential with HTML 5**

Emitting HTML, 2

- ❐ **Since every HTML page starts out the same, we have provided a subroutine, TTFPREB, you can call to generate these first four statements for you**

  - ♦ **It takes no parameters and you just call it, for example:**

```
          call 'ttfpreb'
```

- ❐ **This saves the time and coding to get your basic HTML page starting lines out of the way**

  - ♦ **It also allows us to encapsulate the location-specific information in the link to the stylesheet into only one place**

Notes

- ♦ **TTFPREB uses display to write out html**

- ♦ **Because mixing display and calls to printf don't seem to mix when running under Apache, we have also provided subroutine TTFPREC, which uses printf to write out html**

  - ✗ Call TTFPREC with the same syntax as for TTFPREB above

  - ✗ For class labs, we only use this routine in one place, but you may find a use for it elsewhere

Emitting HTML, 3

❒ As a minimum, you will want to have some lines like these defined (if you are using display, you could just put out literals, but there is some sense in using data items there, too):

```
01  page-title    pic x(47) value
         '<title>Display Environment variables </title>'.
01  head-end pic x(08) value '</head>'.
01  body-start pic x(07) value '<body>'.
01  h2-tag   pic x(40) value
          '<h2>COBOL - Standard CGI variables</h2>'.
01  br-tag   pic x(05) value '<br>'.
01  body-end pic x(08) value '</body>'.
01  html-end pic x(08) value '</html>'.
```

Notes

- If you will be using bpx1wrt for output, each item will need to be a structure, in order to provide the new-line character, such as:

```
01  page-title.
    02  pic x(37) value
         '<title>Display Environment variables </title>'.
    02  pic x  value x'15'.
```

- And if you will be using printf for output, the x'15' should be x'1500', in order to provide null-termination of the strings, for example:

```
    02  pic 02  value x'1500'.
```

Emitting HTML, 4

❒ **Putting out lines using** display **would look something like this**

```
          display page-title
          display head-end
          display body-start
          display h2-tag
```

- **Putting out the same lines using** printf **requires using z in front of the value clauses, then:**

```
          call 'printf' using page-title
          call 'printf' using head-end
          call 'printf' using body-start
          call 'printf' using h2-tag
```

- **The same work using bbx1wrt would be:**

```
       set  buffer-ptr to address of page-title
       move length of page-title   to num-bytes
       perform bpx1wrt-write

       set  buffer-ptr to address of head-end
       move length of head-end     to num-bytes
       perform bpx1wrt-write

       set  buffer-ptr to address of body-start
       move length of body-start   to num-bytes
       perform bpx1wrt-write

       set  buffer-ptr to address of h2-tag
       move length of h2-tag      to num-bytes
       perform bpx1wrt-write
 .  .  .
 bpx1wrt-write.
       call 'bpx1wrt' using stdout,buffer-ptr,
                buffer-alet,num-bytes,return-val,
                return-co,reason-co.
```

Accessing Environment Variables

- ❒ **A simple redirect response is not very interesting: we only write out HTTP headers, not even any HTML**
  - ♦ **In the next section we explore more complex requests, focusing there on GET requests**
- ❒ **In order to find out what request has been made, a CGI generally needs to access various environment variables**
- ❒ **There are two possible techniques here:**
  - ♦ **Use the LE callable service CEEENV - excellent, but not available before z/OS 1.8, so can be a problem in some environments**
  - ♦ **Call the relevant C function,** getenv **- a viable alternative for all releases and compiled languages**
- ❒ **In this course we will demonstrate both of these approaches**

Accessing Environment Variables - CEEENV

❒ **All LE-conforming languages may call the CEEENV service (introduced in z/OS 1.8)**

Generic syntax

CEEENV *request*, *name_len*, *name*, *val_len*, *value*, *fc*

Input	*request*:	**a(fullword binary); "1" indicates "locate value"**
Input	*name_len*:	**a(fullword binary containing length of variable name)**
Input	*name*:	**a(string containing variable name) (not null-temrinated)**
Output	*val_len*:	**a(fullword where length of value is returned)**
Output	*value*:	**a(string containing the value)**
Output	*fc*:	**a(12 byte feedback code area)**

Example

```
   call  'ceeenv' using f1, nL, vName, vL, vValue, fc
```

- **Check fc afterwards, to ensure the call was successful (if fc contains low-values, all went well)**

Accessing Environment Variables - CEEENV, 2

❐ **So, to flesh it out a little in the style we have been working with ...**

- **Our data areas would look something like this:**

```
01  fc       pic x(12) value low-values.
01  f1       pic s9(8) binary value 1.
01  nL       pic s9(8) binary.
01  varname1 pic x(12) value 'QUERY_STRING'.
01  vName    pointer.
01  vValue   pointer.
01  vL       pic s9(8) binary value 0.
```

- **Then the prep and call might look like this:**

```
     set vName to address of varname1
     move length of varname1 to nL
     move low-values to fc
     call  'ceeenv' using f1, nL, vName, vL, Value, fc
     if fc = low-values
        continue
     else
        perform no-val
        goback
     end-if
*   if get here, vValue contains address of string
*   and vL contains length of string
```

- **The use of "vValue" was introduced because "value" is a COBOL reserved word**

    ✗ Then we used "vName" and "vL' to continue the naming pattern

Accessing Environment Variables - getenv

❒ **The** getenv **C function takes as input a null-terminated string containing the name of the environment variable you are interested in**

- **And returns either the address of the null-terminated string containing the value of the variable, or binary zeros if the variable does not exist**

- **So we would set up the variable name and return field as:**

```
   01  Envar-related-variables.
       02 var-name    pic x(13)  value z'QUERY_STRING'.
       02 env-ptr     pointer.
       02 err-ind     redefines env-ptr pic s9(8) binary.
```

- **And call the function this way:**

```
     call  'getenv' using var-name returning env-ptr
     if err-ind = 0
* if get here, issue 'variable not set' message
     end-if
* if you get here, env-ptr contains the address of the
* null-terminated value of the environment variable
```

❒ **Now, let's take a look at how we might display the value we've found - or how to deal with a variable with no value (which means the variable has not been defined)**

- **Our approach is to simply write out some HTML**

  ✗ We will demonstrate using printf, bpx1wrt, and display

Displaying Environment Variables

❒ **Let's suppose for a minute that you are only interested in displaying the value in an environment variable**

- ♦ **Which could be the case during development, debugging, or our next lab(!)**

❒ **In COBOL, we can use** printf, bpx1wrt**, or** display **to emit the value of a variable, regardless of how we got to the value**

- ♦ **However, we will assume from now on that you have used getenv to access the value in an environment variable, since that is available in older systems and ceeenv is only available in newer systems**
  - ✗ Converting from ceeenv usage to getenv usage is left as an exercise for the student
- ♦ **From now on, in any case, we can assume we have env-ptr pointing to the value of an environment variable**
  - ✗ And that value is a null-terminated string

Displaying Environment Variables Using printf

❒ **To use the** printf **function to display a string, you usually pass a message string which includes a "%s" everywhere you want the function to fill in a string value**

- **Followed by a pointer to a null-terminated string for each %s in your message string (matching is done in order from left to right)**

- **So building on our previous work, we might have this in our working-storage section:**

```
01  var-name     pic x(13)   value z'QUERY_STRING'.
01  var-msg      pic x(20)   value z'%s = %s <br>'.
01  err-msg      pic x(34)
       value z'%s: ** variable not set **<br>'.
```

- **That is, both the variable display message and the error message are HTML with text followed by a break**

  ✗ Since we are using getenv and printf, we need to terminate the strings by null characters (using 'z' literals)

- **Note the %s entries; now if we are successful we issue:**

```
     set  address of var-value to env-ptr
     call 'printf' using var_msg, var-name, var-value
```

- **and if we are unsuccessful we could do:**

```
     call 'printf' using err-msg, varname
```

❒ **Note that** printf **always writes to** stdout

Displaying Environment Variables Using BPX1WRT

❒ **To use bpx1wrt for this task takes a little more work: you must write the message out in pieces, using bpx1wrt for each piece**

- ♦ **For example:**

```
01  disp-msg-1 pic x(15) value 'QUERY_STRING = '.
01  disp-msg-end.
    02  pic x(04) value ' <br>'.
    02  pic x     value x'15'.
01  err-msg    pic x(22) value '** variable not set **'.
01  Envar-related-variables.
    02  var-name    pic x(13)   value z'QUERY_STRING'.
    02  env-ptr   pointer.
    02  err-ind     redefines env-ptr pic s9(8) binary.
    02  len         pic s9(8)   binary value 0.
   .
   .
   .
linkage section.
01  var-value       pic  x(256).
   .
   .
   .
```

- ♦ **NOTE: Here we have deliberately set up for a maximum of 256 characters in a message**

  ✗ The actual value could be longer than 256 bytes, but no damage can be done here, since we are only accessing, not changing, the value, and the value is outside our program

Displaying Environment Variables Using BPX1WRT, 2

```
    set  buffer-ptr to address of disp-msg-1
    move length of disp-msg-1  to num-bytes
    perform bpx1wrt-write

    call 'getenv' using var-name returning env-ptr
    if err-ind = 0
        perform getenv-err
    else

        set  address of var-value to env-ptr

        move 0 to len
        inspect var-value tallying len for
           characters before initial x'00'

        set  buffer-ptr to address of var-value
        move len                   to num-bytes
        perform bpx1wrt-write

    end-if

    set  buffer-ptr to address of disp-msg-end
    move length of disp-msg-end to num-bytes
    perform bpx1wrt-write
```

- **Notice we start by printing the initial part of the message, which contains the variable name we are after**
- **Then we get the value ...**
  - ✗ If successful, we scan to get the length of the value and print the value
  - ✗ If no variable is found, we print the err-msg text
- **Then, in either case, we print the message end (causing a new line to be put out)**

Displaying Environment Variables Using BPX1WRT, 3

❒ If we find a variable with it's value, we could have, as before:

```
   bpx1wrt-write.
          call 'bpx1wrt' using stdout,
                               buffer-ptr,
                               buffer-alet,
                               num-bytes,
                               return-val,
                               return-co,
                               reason-co

         if return-val  =-1
           call 'cee3abd' using return-co, clean-up.
```

- Notice above we have added a check that the write was successful, and if not we abend with a user abend code equal to the return code from bpx1wrt; "clean-up" needs to be defined in working-storage as:

```
01  clean-up             pic s9(9) binary value +1.
```

❒ Now, if a variable is not set when using bpx1wrt, we might have:

```
   getenv-err.
         set  buffer-ptr to address of err-msg
         move length of err-msg     to num-bytes
         perform bpx1wrt-write
         .
```

Displaying Environment Variables Using DISPLAY

❒ **Here are the pieces for using DISPLAY to emit the value of an environment variable:**

```
01  Envar-related-variables.
    02  var-name    pic x(13)   value z'QUERY_STRING'.
    02  env-ptr   pointer.
    02  err-ind     redefines env-ptr pic s9(8) binary.
    02  len         pic s9(8)   binary value 0.
   .
   .
   .
linkage section.
01  var-value       pic  x(256).
   .
   .
   .
     call 'getenv' using var-name returning env-ptr
     if err-ind = 0
          display 'QUERY_STRING = '
                  '** variable not set **<br>'
     else
         set  address of var-value to env-ptr
         move 0 to len
         inspect var-value tallying len for
            characters before initial x'00'
         display 'QUERY_STRING = '
                 var-value(1:len) '<br>'
     end-if
```

Writing to stderr

❒ When a message really should go to stderr instead of stdout, you have two choices, again:

- Use the LE CEEMOUT callable service
- Use bpx1wrt with a routing to stderr (fullword '2') instead of stdout (fullword '1')

❒ If you will be using bpx1wrt, you can use the sprintf C function to format a buffer in the same way that printf does its formatting; for example:

```
      call  'sprintf' using work-out, err-msg1, var-name
      set  buffer-ptr to address of work-out
      move 0 to len
      inspect work-out tallying len for
         characters before initial x'00'

      set  buffer-ptr to address of work-out
      move len                   to num-bytes
      call 'bpx1wrt' using stderr,
            buffer-ptr, buffer-alet,
            num-bytes,  return-val,
            return-co,  reason-co
```

- Using these new data item definitions:

```
01  work-out     pic x(256) value spaces.
01  stderr       pic s9(8) binary value 2.
01  err-msg1.
    02 pic x(34)
          value '%s: ** variable not set **'.
    02 pic x value x'15'.
```

Writing to stderr, 2

❒ **Lines written to *stderr*, however, end up in the cgi-error log for the HTTP server, not always easy to get to**

❒ **Note that you can mix and match the use of printf, getenv, ceeenv, strlen, bpx1wrt, sprintf, ceemout all in the same program, as needed**

- ♦ **You only need to be aware of the formats of arguments**

bpx1wrt vs. printf() vs. DISPLAY

❒ **As with most techniques in programming, these approaches for writing to** stdout **each have their own pros and cons**

bpx1wrt

- **This is a z/OS UNIX kernel command; not dependent on C-specific interfaces**
- **More arguments (so more set up)**
- **Strings must not be null-terminated**
- **May build up a line by bpx1wrt-ing each piece separately; required if you intend to format one or more of the pieces**
- **May use to write to** stderr **also**

printf()

- **C-specific**
- **Fewer arguments**
- **String arguments (both in the message string and in any strings passed to match up to %s formats) must be null-terminated**
- **Formating is done based on your arguments and format indicators**
  - ✗ So must have all the pieces in place before calling printf()
- **Always writes to** stdout

display

- **COBOL-specific**
- **Always writes to stdout**
- **More work to display null-terminated strings**
- **More work than printf to format data**

Stylesheets and CGIs

- ❐ Generally, a static HTML page is served from a particular directory, and CGIs are run out of a different directory

- ❐ When a CGI references a stylesheet and it is a relative reference (for example: <link ... href="cgi-style1.css" type="text/css">), the stylesheet is presumed to be found in the CGI directory

  - ♦ But because of the configuration values normally set up, files found in the CGI directory are presumed to be executables and the server tries to run the stylesheet instead of just pass it on to the server

Stylesheets and CGIs, 2

❒ **To fix this, you need to provide a mapping in your configuration files, and there are several ways to go, including these two:**

- ♦ **Create a string that maps to a common, shared directory for styles; here's an example:**

```
Pass   /t-css/*   /usr/lpp/testing/*
```

  - ✗ Then in your CGI your link to a stylesheet might be:

```
<link ... href="/t-css/cgi-style1.css" ... >
```

- ♦ **Create a string that maps to one of your directories, maybe even your CGI directory; for example:**

```
Pass   /s-css/*   /u/scomsto/CGI/*
```

  - ✗ Then in your CGI your link to a stylesheet might be:

```
<link ... href="/s-css/cgi-style1.css" ... >
```

❒ **Of course, this is normally done by a systems person, and not lightly because it requires recycling the HTTP server**

- ♦ **So you build one mapping per person, and have each person work in their own directory   or**

- ♦ **You build a single, shared mapping and everyone uses a shared directory   or a combination**

Computer Exercise: Writing Out HTML Pages

In this exercise we work on displaying some environment variable values, and laying the base for our future work. All the source code here is in your TR.COBOL library. To compile and bind, use member COBCGIA in your TR.CNTL library, just change the value of the SET O= line to point to the code to work with.

The source code to work with:

TTFPREB - writes out the first HTML headers, as discussed on page 27
TTFPREC - writes out HTML headers using printf, see page 27

TCBVARP - uses printf for writing to stdout
TCBVARB - uses bpx1wrt for writing to stdout
TCBVARD - uses display for writing to stdout

The last three programs call TTFPREB (well, TCBVARP uses TTFPREC) and output a page that displays the values of the environment variables QUERY_STRING and SERVER_SOFTWARE.

Your tasks:

Change TTFPREB and TTFPREC, fixing up the system name and userid then compile and bind this program.

Change TCBVARP or TCBVARB or TCBVARD (or more than one, if you are so inclined) to add displays of the contents of REMOTE_ADDR and REMOTE_USER. Compile and bind.

Deploy TCBVAR*x*, as discussed earlier (see page 16).

Test your work by pointing your browser to CGI_Labs.html:

* Select option 2, which takes you to test_display_cob.html

* This page asks for a userid and password (you can use anything, as they are not checked - yet) and for you to select the name of the CGI you want to test; Fill these items in and select Submit; you should see the output from your CGI. Test all the CGIs you have prepared.

Take some time and study the outputs, especially for QUERY_STRING.