

# Coding for speed, size and portability

## Contents

- **Structure**
  - Program structure
  - Application structure
- **Data**
- **Code**
- **Directives**
- **Appendices**

## Structure

### Program Structure

Otherwise known as control flow

Your program should be considered as a collection of logically separate procedures. Each procedure should be commented with respect to its input and output and function. There is no trickle through - instead each procedure is invoked using perform.

The section is the procedure, the section name the procedure name. If you like, think of sections as Pascal procedures, or C void functions.

Paragraphs are internal labels that should be regarded as not visible outside the section in which they are used. Perform is the function/procedure call.

To achieve this model there are some easy rules.

|             |  |
|-------------|--|
| Do use      | goback at end of first section - must be inline code<br>perform section  |
| Do NOT use  | alter<br>perform thru<br>next sentence<br>segments<br>go to section-name<br>performparagraph<br>go toparagraph outside current section |
| You may use | go to paragraph within a section.  |

Programs that follow these guidelines are known as well-behaved, or (well) structured. Well-behaved programs are faster to generate, produce smaller and faster code, and are easier to understand. A win-win-win situation.

It is vitally important to use goback [ returning ... ] or stop run as the last statement of the first section. This makes it impossible for control to trickle into subsequent sections, which allows the native code generator (ncg) to generate smaller and faster code for performs.

Other miscellaneous control flow guidelines are

### *goto depending*

Fast, efficient, but can be confusing. Ensure all targets are within the section

### *evaluate*

Do not use an expression in the evaluate clause. Use a temporary. *when* clauses should be ordered so that the most likely cases come first.

### Application structure

## Coding for speed, size and portability

This section is couched in UNIX terms, but the underlying concepts are cross-platform.

.gnt code is loaded into the data section and cannot be shared between processes (the same program running more than once). Therefore, if 2 or more copies of the program may be running simultaneously, build to executable. If you have a suite of applications, where many application's share some common sub-programs then the situation is more complicated.

Let us consider the simplest case where 2 applications each have a separate main module, namely "a" and "b", and a common module "sub". a and b are both being run twice.

If these were all in .gnt code, then 2 each of a.gnt and b.gnt are loaded and 4 copies of sub.gnt - substantially reducing system performance. .lbr's make no difference.

If we build to executable : eg

```
cob -xO a.cbl sub.cbl ; cob -xO b.cbl sub.o
```

then there is only 1 copy of a and b loaded, but the code of sub is loaded twice. There are two solutions to this problem

Use dynamic linking. Create a shared object containing sub.o, and 2 dynamically linked executables a and b eg using Server Express

```
cob -xcO a.cbl b.cbl sub.cbl  
cob -Zo libsub.so sub.so  
cob -x a.o libsub.so  
cob -x b.o libsub.so
```

This topic is dealt with in greater detail in the Building\_Unix\_Applications document also issued in the July 2003 SupportLine Newsletter.

Alternatively, use the C program in Appendix A. Create an exe using

```
cob -xO cobcmd.c a.cbl b.cbl sub.cbl  
In a b
```

This creates 1 executable containing all the code. The C program decides from the command line which app to run. Only 1 copy each of a, b and sub is ever loaded. NB : the link between a and b must be preserved.

### **Data**

Align, align, **ALIGN !**

On **ALL** chips, you must align 2 byte data items on 2 byte boundaries, 4 byte items on 4 byte boundaries. Increasingly, it pays to align larger data items on 8 byte boundaries. Misaligned data takes longer to load and store and may require more instructions. When a misaligned item is an operand and the target of arithmetic (ie add a to b) then you get the penalty twice. The situation will only get worse with new chips and new versions of existing chips

| Chip            | Cycles to store pic x(4)<br>comp-5 |            | Additional Instructions for misaligned |
|-----------------|------------------------------------|------------|--|
|                 | aligned                            | misaligned |  |
| Pentium         | 1                                  | 4          |  |
| RS6000, PowerPC | 1                                  | 2          |  |
| MIPS            | 1                                  | 2          | 1                                      |
| HP PA           | 1                                  | 7          | 6                                      |
| Sparc, Alpha    | 1                                  | 10         | 9                                      |

## Coding for speed, size and portability

|         |   |   |   |
|---------|---|---|---|
| Itanium | 1 | 4 | 8 |
|---------|---|---|---|

### Arrays

An array should have a stride (ie total length of 1 table item) that is a multiple of 4 bytes. This means the alignment of all the subscripted items can be calculated. Ideally the stride is a power of 2.

### Arithmetic data

*comp* is slow on all lohi platforms. This includes all Windows, OS2, and DOS platforms. The bytes need to be swapped. Use comp-5 for speed - comp-x for portable data files. Use 1, 2 or 4 byte items. Use integers.

### Alphanumeric

Alphanumeric items should be a multiple of 4 or 8 bytes where possible.

See Appendix C for the results of misalignment.

### Code

#### Arithmetic

Try to use unsigned comp-5 items that are the same size. Add and subtracts can have multiple sources, targets and giving clauses (beware of subscripted items though). Add and subtract are always optimized in computes. The optimal length is 4 bytes. Try to avoid conversions to/from display and comp-3 items.

Comparisms against the literal zero are usually the most efficient. Comparisms for (in)equality are usually more efficient than compare for ordering, particularly on lohi machines (ie Pentium).

Multiply is expensive - divide is exorbitant - divide remainder is prohibitive. You have been warned. Multiply by small literal values however is usually reasonable. Multiplication/division by powers of 2 is optimized.

The COBOL rules on the precision of intermediate results are why 2 operand statements are in general better. In particular

```
if a + b > c
must be evaluated for 5 bytes (in effect 8) assuming that either a or b is 4 bytes.      evaluate a
+ b
is evaluated for every single when clause. Use
compute c = a + b
evaluate c
```

### Alphanumerics

Comparisms for (in)equality are faster than comparing for ordering, especially on lohi chips such as Pentium. Alphanumeric items should be multiples of 4 bytes long, and be 4 byte aligned. Variable length moves (ie reference modified, where the length is a variable) are slower than fixed length as they must be done byte by byte. Also - there is no possibility of loop-unrolling and an additional check must be done for 0 length moves.

### Calls and files and environment variables

Be consistent in the case you use for calls, filenames and environment variables. UNIX and C are case-sensitive. Lower case is the norm for filenames and upper case for environment names. Some

## Coding for speed, size and portability

modules are in upper case, such as PANELS and FHREDIR. Be sure to use the correct case. Use call literal in preference to call data name. The same applies to *set procedure-pointer to entry*.

Do not include extensions or paths - use \$COBPATH if files are not in the same directory.

Do not use PC\_ calls or call by number routines - use the appropriate CBL\_ routines which also use comp-5. This gives portability and better performance on all platforms.

*call on overflow* implies that the called program may not be there. Therefore in .o (.obj) a static reference cannot be produced. If an on overflow clause is there only for .int/gnt environments, and you know the modules will be linked then use call convention 8.

Call parameters should be 01 level items, and subprograms generated *Inkalign* Do not rely on knowing how many parameters have been passed to subprogram. This is very expensive. If different sub-functions have a different number of parameters, determine this from a sub-function argument.

### Entry points and parameters

Parameter processing is most efficient when

1. program is generated notypecheck nolnkcheck noparamcountcheck. Better still - use nocheck.
2. There is only 1 entry point or parameter lists do not conflict.

A non-conflicting entry list is a proper prefix of every longer parameter list in the same program.

Such programs are known as fastlink. In fastlink programs, by value accesses are optimized. Otherwise by value is no faster than by reference (and actually has a slightly higher overhead during program initialization).

A fastlink program cannot use

*if address of <parameter> = NULL*

which requires *paramcountcheck*

If you absolutely have to use this syntax, then any call that does not supply this parameter should pass by value NULL instead.

Do not use by value parameters > 4 bytes. It is not portable.

### Coding style

This is all subjective of course - but this works.

1. Use all scope-delimiters. Enforce by *\$ set noimplicitscope change-message(1227 E)*

The ANS85 scope-delimiters are optional. A construct, eg if, nested inside another, ie evaluate, is terminated by end-evaluate or when. This is confusing and potentially buggy. Use the delimiters.

2. Only use a period ('.') to delimit paragraphs and sections.  
'.' can be the source of many and subtle and devious bugs. How many times has an extra "." caused a compilation error (if you used scope delimiters) or worse - a run time bug (if

## Coding for speed, size and portability

you did not). Have you ever added code to the end of the section only to get a compilation error because you forgot the '.'? I use

- o . process-line section.
- o . proc-line-loop.

to declare sections and paragraphs. This keeps all the '.' on one line, and neatly delineates the section name

3. Avoid goto by using
  - o in-line performs
  - o exit section
  - o exit perform
  - o exit perform cycle

(see Appendix B)

4. Replace multiple else if by evaluate true. Evaluate true is just as efficient as if ... else if. It is more readable, and your code does not inexorably creep across the page and off the right margin.
5. align end-verb with verb, and when with evaluate
6. Use typedefs and call prototypes.  
typedefs and call prototypes are used in almost all languages since ALGOL 60. They enforce good coding style and eliminate a lot of bugs. Enforce adherence to call prototypes by using

*\$set change-message(1056 E 1057 E 1058 E 1059 E)*

If you need to override the prototype for some reason (eg the checker sometimes does not accept value pointer as equivalent to reference use)

*\$set change-message(1057 N)*

### Directives

|         |            |   |
|---------|------------|---|
| checker | noalter    | improves checker speed  |
|         | noseg      | improves checker speed  |
|         | noqualproc | improves checker speed<br>catch identical and confusing paragraph names |
|         | nocheck    | turn off run time checking  |
| nccg    | nocheck    | turn off run time checking  |
|         | opt        | turn on global optimizations  |
|         | lnkalign   | guarantees linkage items are aligned                                    |

### Appendices

#### Appendix A - cobcmd.c

```
#include

main(argc, argv)
    int argc;
    char **argv;
{
    char *commandline;

    commandline = cobcommandline(0, &argc, &argv, NULL, NULL);
    if(commandline == NULL)
    {
        fprintf(stderr, "Cannot create command line!!\n");
    }
}
```

## Coding for speed, size and portability

```
        exit(1);
    }

    if(strcmp(argv[0], "a") == 0)
        a(commandline);

    if(strcmp(argv[0], "b") == 0)
        b(commandline);

    fprintf(stderr, "Unknown application name : %s\n", argv[0]);
    exit(1);
}
```

### Appendix B : gotopfn.cbl

```
        $ set noqualproc noseq noalter sourceformat(free)

01  a      pic x comp-5.
    88  condition          value 0.

*> Three different ways to code the same loop. We want to accurately
*> reflect the real structure of the algorithm, avoid gotos, and
*> avoid repetition of code.

procedure division
. main section.
    perform loop1
    perform loop2
    perform loop3
    stop run

*> "go to"s and "if"s everywhere and code repeated - almost unintelligible

. loop1 section.
    *> block_a
. para-a.
    if condition
        go to para-b
    end-if
    *> block_b
    *> block_a
    go to para-a
. para-b.
    exit

*> Much better - got rid of the repetition and "go to"s

. loop2 section.
    perform block-a
    perform with test before
        until condition
            *> block_b
    perform block-a
    end-perform

*> The best - No out of line performs, overall structure correctly
*> represented

. loop3 section.
    perform until exit
        *> block_a
        if condition
            exit perform
    end-if
    *> block_b
```

## Coding for speed, size and portability

```
end-perform
. block-a section.
  *> block_a
```

### Appendix C : misalign.cbl

```
    $ set noalter noqualproc noseq sourceformat(free)

01 occurs 20.
*> Stride of 7 - cannot know alignment of item with variable subscript
  03 a1 pic x(4) comp-5.
  03 b1 pic x(3).
01 occurs 20.
*> Stride of 8 - items always aligned
  03 a2 pic x(4) comp-5.
  03 b2 pic x(3).
  03          pic x.

01.
  03 src          pic x(24).      *> aligned
  03 tgt2         pic x(24).      *> aligned
  03          pic x.
  03      tgt1    pic x(24).      *> misaligned

01.
  03          pic x.
03 i1          pic xx comp-5.
03 j1          pic xx comp-5.
01.
  03 i2 pic xx comp-5.
  03 j2 pic xx comp-5.

procedure division
. main section.

*> Figures quoted are for HP's Precision Architecture chip

. para-1.
  *> 33 instructions, 33 cycles
  add a1(i1) to a1(j1)

. para-2.
  *> 8 instructions, 8 cycles
  add a2(i2) to a2(j2)

. para-3.
  *> 10 instructions, 59 cycles !
  move src to tgt1

. para-4.
  *> 8 instructions, 8 cycles
  move src to tgt2

. final-para.
  stop run
```