Applying the ABC Metric to C, C++, and Java

Jerry Fitzpatrick

Metrics are becoming increasingly important in software design and analysis. Experience shows that even simple metrics, when properly applied, provide valuable insights into the software development process.

The ABC metric provides a measure of software size. A size measure does not provide sufficient design guidance by itself, but it has several uses and is an underpinning for more advanced measure.

I developed the ABC metric to overcome the disadvantages of lines of code and similar measures. Before examining the benefits of the ABC metric, let's review some other software size measures.

Size Measures

There are many potential ways that software size can be measured. Regardless of how measurements are made, an ideal metric should be:

- measurable
- accountable
- precise
- independent of the measurer

One of the earliest methods of measuring software size was developed in 1975 by the late Maurice Halstead of Purdue University[1]. Using an approach dubbed "software science" Halstead reasoned that the syntactic components (tokens) of every computer program fall into two distinct categories: operators and operands.

An operator is a token – like plus, equals, or function call. An operand is any token that is not an operator. By counting the number unique operators, operands, and their combined total, software science formulas provide numeric values for program length, volume, and level.

Although software science appears scholarly and intriguing, its conclusions have been widely disputed. Consequently, token counting has not been widely adopted. Halstead's contribution is nevertheless important because he applied analytical techniques to the then young field of software development.

The oldest and most popular way to measure a program's size is by lines of code (LOC). Two reasons LOC is popular are that it is easy to calculate, and most developers have an intuition as to what a line of code is. But the LOC metric has never been formalized by any standards organization, and there are several variations in the methods used for counting. However, in the core analysis, a line of code is considered to be a nonblank, non-comment line of program code.

Although program size if often quoted in LOC, this system of measurement has many shortcomings. To begin with, the LOC measure has no theoretical foundation, as there is no clear relationship between lines of code and program operation.

Also, LOC is a very unreliable measure. Consider the code fragments shown in Figures 1(a) and (b).

Although these routines are functionally and semantically identical, their LOC value varies by a factor of three! This example demonstrates that the LOC count is dependent on programming style. This dependency implies that LOC may provide inconsistent results between projects and programmers. This inconsistency may mean that our previously-mentioned "developer's intuition" regarding lines of code may not be very good.

A different type of size metric that has become increasingly popular is called "function points" (FP). LOC and the software science formulations are *code metrics*, unlike FP, which is a design metric.

Originally published in C++ Report, June 1997.

```
bool SearchList(char *cmd, char **list, unsigned n) {

    // search entire list for command string
    for(unsigned i=0; i < n; i++)
        if (stricmp(cmd, list[i]) == 0) return true;

    return false;
}
```

Figure 1(a) – A Concise Style (LOC = 5)

```
bool SearchList(
    char *cmd,          // command string
    char **list,        // array of strings
    unsigned n)         // max. no. of elements
{
    unsigned i;

    // search entire list for command string

    for(unsigned i=0; i < n; i++)         // for each element
    {
        if (stricmp(cmd, list[i]) == 0)       // matched?
        {
            return true;                  // found match
        }
    }

    return false;                         // no match
}
```

Figure 1(b) – A Verbose Style (LOC = 15)

Function points were introduced in a paper published in 1979 by Alan Albrecht, then of IBM. FP is based on the concept that the workings of a computer program are determined by the data generated and retained by that program. The number of function points in an application is the weighted sum of the program inputs, outputs, files, and inquiries.

Contrary to the terminology, function point calculations do not directly involve program function (routines); nor do they relate to the specific functionality of the program. Instead, the term *function* is intended to denote the program's capability for transmuting data and user input and output.

It is important to understand that the FP metric provides a high-level program size measurement. Its primary advantage is that a program's size can be estimated from specifications before it is written. FP is not an appropriate measure for small modules, such as individual files or routines, and applying it at this level often leads to counter-intuitive results.

The ABC metric was developed to overcome the disadvantages of LOC and token-counting techniques. However, since it is a code metric it supplements, rather than replaces, measures such as FP.

ABC Calculations

Imperative programming languages like C, PASCAL, and other common languages, use data storage (variables) to perform useful work. Such languages have only three fundamental operations: storage, test (conditions), and branching.

A raw ABC value is a vector that has three components:

- Assignment – an explicit transfer of data into a variable
- Branch – an explicit forward program branch out of scope

- Condition – a logical/Boolean test

Each component identifies the number of fundamental operations. Data storage usually takes the form of assignment – the A in ABC. Branching (the B in ABC) usually takes the form of function calls. Finally, conditions (the C in ABC) are provided by *if*, *else*, and similar constructs. The complete definitions for each component of the ABC metric follow.

There are many other aspects of a program that could be factored into a size measure. For example, you could consider variable declarations, arithmetic operators, and so on. Many of these factors are important to program complexity, but not necessarily to program size.

Metric design is somewhat like language design, as it requires reasoning, experimentation, and a number of tradeoffs. Although it is not entirely possible to distinguish between size and complexity, my goal for the ABC metric was only to isolate the most fundamental components of size.

The FP metric was developed with data-processing systems in mind, but it has recently been applied to more diverse systems. The International Function Point Users Group (IFPUG) has provided standardization and popularized the technique.

ABC values are written as an ordered triplet of integers. For example, a subroutine may have an ABC value of <7,12,2> where 7 is the assignment count, 12 is the branch count, and 2 is the condition count. To standardize the notation, the counts are always written in the order A, B, and C, and are enclosed by angle brackets.

To simplify code comparisons it is necessary to use a scalar size measure. However, because each component of an ABC vector is a distinct entity, it is not feasible to add them together to form a single value.

Instead, I have assumed that the components are orthogonal and have comparable scales. On this basis, a single ABC size value may be obtained by finding the magnitude of the ABC vector using the following formula:

```
|ABC| = sqrt((A*A)+(B*B)+(C*C))
```

Applying this calculation to the subroutine mentioned in the previous paragraph yields an ABC magnitude value of 14.0.

Further research may show that the scales are not comparable and that a weighted sum or other calculation will provide better results. For now, however, I am recommending the magnitude calculation.

An ABC magnitude value is always rounded to the nearest tenth and is reported using one digit following the decimal point. Also, ABC magnitudes should not be presented without the accompanying ABC vector.

Because of semantic differences between programming languages, the specific rules for counting ABC values must be interpreted within the context of that language. Instead of covering more background theory, let's look at the ABC rules for C, C++, and Java which have been formulated to provide comparisons between the languages.

C Rules and Examples

Let's begin by examining the counting rules for C (see Figure 2), the predecessor of C++ and Java. Note that although the counting rules may be used as stated, there may be alternative rules that achieve the same values.

1. Add one to the assignment count for each occurrence of an assignment operator (excluding constant declarations):

   ```
   =  *=  /=  %=  +=  <<=  >>=  &=  |=  ^=
   ```

2. Add one to the assignment count for each occurrence of an increment or decrement operator (prefix or postfix):

   ```
   ++  --
   ```

3. Add one to the branch count for each function call.

4. Add one to the branch count for any goto statement which has a target at a deeper level of nesting than the level of the goto.

5. Add one to the condition count for each use of a conditional operator:

   ```
   ==  !=  <=  >=  <  >
   ```

6. Add one to the condition count for each use of the following keywords:

   ```
   else  case  default  ?
   ```

7. Add one to the condition count for each unary conditional expression.

Figure 2 – ABC Counting Rules for C

In many ways, the C language is simpler than the C++ or Java languages. Although a few comments are in order, the seven counting rules are self-explanatory.

Rule 1 specifically excludes constant declarations such as:

```
const double PI = 3.14159;
```

The reason for this exclusions is the `PI` is not a variable and only variable assignments (i.e. data storage) are counted. Although the syntax is almost identical to a normal assignment, the expression joins the symbol `PI` and the value 3.14159.

Rule 2 adds increment and decrement operations to the A count because they implicitly perform an assignment. For example, the expression:

```
x++;
```

is equivalent to:

```
x = x + 1;
```

In the same way, Rule 6 states that a ternary expression such as:

```
x = (a == 3) ? a : b;
```

is equivalent to:

```
if (a == 3)
```

```
   x = a;
else
   x = b;
```

Usually, according to the ABC definitions, only explicit assignments are counted. But by treating these equivalent but semantically different expressions in the same way, the ABC count provides a greater consistency between programming styles and languages.

Function calls increment the branch count, but function returns are not counted, as their direction is considered to be backward, not forward.

Rule 4 says that certain **goto** statements are also branches. These are tricky to determine automatically. Fortunately, there aren't usually many **goto**s in a well-structured program.

Lastly, the unary conditional expression in Rule 7 is an implicit condition that uses no relational operators. In the construct:

```
if (x || y)
   printf("test failure\n");
```

there are two unary conditions since both x and y are tested as conditional expressions.

Listing 1 shows the ABC count for a small C subroutine. The annotations to the left of each line of source code show where each component of the ABC count is generated. The ABC vector for this routine is <5,6,3> and has a magnitude of 8.4.

A more complicated C routine is shown in Listing 2. In this example the ABC vector is <7,12,14> and it has a magnitude of 19.7.You should notice that it becomes much easier to make a mistake when performing an ABC count for larger routines.

There are no symbolic constants in this routine because it has been run through the C preprocessor to expand macro definitions. To ensure accuracy, the ABC counting rules should only be applied to C code that has been preprocessed. We will examine the reason for this in the C++ example below.

C++ Rules and Example

The ABC counting rules for C++ (see Figure 3) are similar to the counting rules for C. However, there are some important differences.

As with C, constant declarations are not counted. Similarly, default parameter assignments such as:

```
class Point
{
    Point(int x=0, int y=0);
};
```

are not counted.

The **try** and **catch** keywords for C++ exception-handling are additional conditionals not found in C.

Branches occur not only for free function calls, but also for class method calls (see Rule 4). The new and delete operators are counted as function calls (see Rule 6) mainly because they are equivalent to the function calls **malloc** and **free**.

You may wonder why operators are not counted as branches, since their operation is often identical to a function call. For example, the statement:

```
x = y + 6;
```

whose ABC = <1,0,0> could be rewritten as:

```
x = Add(y,6);
```

whose ABC = <1,1,0>.

It is important that the rules for counting are made in the context of the language and not the underlying implementation. The definition of an ABC branch is an explicit forward program

branch out of scope. In the original expression, we do not know, and should not consider, the implementation of the addition operator. Even though we may know that the expression is merely "syntactic sugar", the operation is still not considered to be an ABC branch.

1. Add one to the assignment count for each occurrence of an assignment operator. Be sure to exclude constant declarations and default parameter assignments:

   `=  *=  /=  %=  +=  <<=  >>=  &=  |=  ^=`

2. Add one to the assignment count for each occurrence of an increment or decrement operator (prefix or postfix):

   `++  --`

3. Add one to the assignment count for each variable and nonconstant class member initialization.

4. Add one to the branch count for each function call or class method call.

5. Add one to the branch count for any goto statement whose target is at at a deeper level of nesting than the goto.

6. Add one to the branch count for each occurrence of the **`new`** and **`delete`** operators.

7. Add one to the condition count for each use of a conditional operator:

   `==  !=  <=  >=  <  >`

8. Add one to the condition count for each use of the following keywords:

   `else  case  default  try  catch  ?`

9. Add one to the condition count for each unary conditional expression.

Figure 3 – ABC Counting Rules for C++

Unlike C and Java, C++ has a special initialization syntax that can be more efficient than traditional assignment. However, initialization still amounts to data storage, so non-constant initializations are counted as an ABC assignment.

Each of the two C++ constructors shown below have the same ABC vector <2,0,0>:

```
List::List(void)
{
    itsData = 0;
    itsElements = 0;
}
```

```
List::List(void) : itsData(0), itsElements(0)
{
}
```

Listing 3 demonstrates how the ABC metric is applied to a simple C++ class method. The ABC vector for this method is <5,13,1> and the magnitude is 14.0.

As in previous C examples, this C++ code has been preprocessed. Preprocessing is necessary because macros and preprocessor directives may hide elements of the program that are needed for an accurate ABC count. Consider the following program:

```
#include <stdio.h>

#define EQUALS     =
#define LT         <
#define GT         >
#define INC(x)     ++(x)
#define DEC(x)     --(x)

void main()
{
    int y;
    for (y EQUALS 0; y LT 10; INC(y))
        printf("%u %u\n", y, y*y);
}
```

Although this is a bit unusual, it is not uncommon for beginning C/C++ developers to create macros reminiscent of another language they may be more familiar with.

Determining the ABC by hand for this unprocessed program is not very difficult, but it would be easy to make a mistake. However, because the semantics are nonstandard, an automated program could not parse this code directly.

The use of preprocessor directives is another problem. Debug statements may not always be removed from released code. One way to eliminate a debug statement is through the use of comments, such as:

```
for(i=0; i < n; i++)
{
    outportb(0x378, data[i]);
    // delay(100);     // slow is down (debug only)
}
```

Yet another way is to use a preprocessor directive in this manner:

```
for(i=0; i < n; i++)
{
    outportb(0x378, data[i]);

#if 0
    delay(100);        // slow is down (debug only)
#endif
}
```

Without preprocessing it's difficult for an ABC counting program to ignore these unused sections of code. Although preprocessing is a nuisance in measuring C and C++ programs, it is not needed in Java programs.

Java Rules and Example

Java does not use header files or a preprocessor like C and C++. This simplifies the language in many ways, but trades off some expressive power by doing so. However it does make ABC counting easier.

Figure 4 shows the ABC counting rules for Java. The rules are similar to, and simpler than, the counting rules for C++.

Java has one extra assignment operator that C++ doesn't; the `>>>=` operator. The `>>>=` operator performs an unsigned right shift and assignment, whereas the `>>=` operator performs a signed shift and assignment.

As in C and C++, constant assignments are not counted. However, constant expressions are identified by the `final` modifier rather than the `const` modifier.

Finally, `goto`s are not implemented in Java, which provides one less branch counting rule for the ABC metric. However, Java reserves the `goto` as a keyword to provide better error-checking at compile time.

1. Add one to the assignment count for each occurrence of an assignment operator, excluding constant declarations:

   ```
   =  *=  /=  %=  +=  <<=  >>=  &=  |=  ^=  >>>=
   ```

2. Add one to the assignment count for each occurrence of an increment or decrement operator (prefix or postfix):

   ```
   ++  --
   ```

3. Add one to the branch count for each function call or class method call.

4. Add one to the branch count for each occurrence of the `new` operator.

5. Add one to the condition count for each use of a conditional operator:

   ```
   ==  !=  <=  >=  <  >
   ```

6. Add one to the condition count for each use of the following keywords:

   ```
   else  case  default  try  catch  ?
   ```

7. Add one to the condition count for each unary conditional expression.

Figure 4 – ABC Counting Rules for Java

Listing 4 shows a simple Java class. When compiled, the class forms a complete application that displays the contents of a file in hexadecimal and ASCII form.

Notice that the constant `FIELD_WIDTH` and `HEXADECIMAL` have been excluded from the count. Also notice that the statement:

```
s = Integer.toHexString(tmp).toUpperCase();
```

counts as two branches since two class methods are called.

Because there is only one class method (main) in the dump class, the ABC vector is applicable to the entire class. If more than one method were implemented, the ABC vector for the class would be equal to the sum of the vectors of each method.

Applications of the ABC Metric

My original interest in source code metrics was to gain a better understanding of modules and their relationships. It is almost impossible to compare the amount of code in two modules without an accurate size measure.

Existing size measures did not seem to me to capture the essence of software size reliably or practically. Because the ABC metric provides useful, style-independent measurements, I'm advocating its use in place of LOC for both private and published studies. In most cases, the ABC magnitude is best for estimations.

One application of the metric is for estimating project schedules. The first step is to estimate the number of ABC's in the completely project. This number is divided by the average number of ABC's the developers can produce per day. The resulting estimate is the number of days to complete the project.

As with LOC usage, the overall project estimate is the most subjective step and is therefore easily misjudged. Practice makes perfect, however, and this technique should still provide reasonable time estimates for small projects. Better yet, the progress of development may be tracked by monitoring the ABC counts on a daily or weekly basis.

Software reliability is often quoted in terms of bugs/LOC and could easily be replaced by bugs/ABC. Of course, either king of macroscopic measurement implies that all parts of the code have an equal potential for bugs – a notion not clearly collaborated by any studies.

As we're seen, the LOC metric is made unreliable by its dependency on a specific style. As a result, project estimates and reliability figures are apt to be inaccurate from project to project. By using ABC magnitudes instead of LOC values, the accuracy of these figures should improve considerably.

More importantly, the actual productivity (in ABC/day or similar) can be accurately compared between developers, products, and businesses. Because syntax of the languages is similar, the ABC counts for C, C++, and Java should allow direct comparisons of programs written in any of these languages.

There are other potential applications of the ABC metric you may find more interesting.

It is commonly accepted that cohesion is a desirable attribute of functions (subroutines). It is usually assumed that the large a function is, the less likely it is to be cohesive. This seems to make sense, but the exact relationship is not clear. However, by analyzing a large number of programs, it may be possible to statistically identify modules that lack cohesion (and require rework).

An accurate size measure also provides a vehicle for source code visualization. Using size and other relevant code measures, an application could be viewed graphically to enhance the understanding of the architecture and topology. This could provide insights into pathological designs, potential malfunctions, and maintenance concerns.

Although there are many potential applications of the ABC metric, you should keep in mind that it does not provide architectural or complexity information. The metric does not directly utilize comments, macros, data flow, or other aspects of a program. It simply provides a low level, style-independent measure of program size.

Benefits of the ABC Metric

The ABC metric has several properties that make it more useful and reliable than LOC. First, the metric is based on the idea that a program performs "useful work" for which data storage, tests, and branching are the basis of operation. This concept is easily understood by anyone who has written a program using an imperative programming language.

Because ABC counting is based on these basic operations, the metric is virtually independent of a programmer's style. This is an important attribute since styles vary enormously in languages such as C and C++.

Another valuable property of the ABC metric is its linearity. ABC measurements can be given for any module (i.e. any section of code have the same scope), whether it is a subroutine, package, class method, file, and so on.

The ABC vector for any module is the sum of the vectors of all its sub-modules. For example, the ABC vector of a C++ class implementation is the sum of the vectors of each of its class methods.

This linearity makes it easy to compare the sizes of functions, files, and classes with complete accuracy. In essence, the topology of the program can be clearly mapped.

Although ABC vectors are linear, ABC magnitudes are not. This non-linearity, coupled with the assumption that the component scales are comparable, is the primary reason that ABC magnitudes should not be reported without the original vectors. In this way, a change in the way ABC magnitudes are calculated will not undermine published research.

It may also be possible to gain more insight into the nature of an application by retaining the ABC counts in vector form. Tom DeMarco[3] has categorized programs as being "data strong" or "function strong". Viewed this way, programs in which assignment statements are dominant could be considered "data strong". Programs in which branches are dominant could be considered "function strong".

Although DeMarco does not make this distinction, programs in which conditionals are dominant could be considered "logic strong". Accordingly, the ABC metric can provide additional information about the driving principles behind the application.

Currently, there are no guidelines for establishing how much larger one component should be to be considered dominant. Additional research should provide some interesting ratios.

Another interesting property of the ABC metric is that it is zero-based. Consider the following C++ destructor:

```
AbstractPoint::~AbstractPoint(void)
{
    // does nothing
}
```

The LOC value for this destructor is three, which suggests the code is actually performing some action. On the other hand, the ABC vector components and magnitude are all zero, signifying that the destructor does not contribute directly to program size. This does not necessarily imply, however, that the destructor is not needed.

Perhaps the most important aspect of the ABC metric is that it seems to judge a program's size the way that most developers do. For research purposes, as set of C++ class methods was ranked by eighteen developers who had little familiarity with LOC and no familiarity with ABC. Their rankings shows a significantly higher correlation to ABC magnitudes than to LOC[6].

The result is not too surprising since LOC is merely a convenient measure, not a well-constructed one. No one is fooled into thinking that there is a greater amount of source code if it is spread out over several lines. Likewise, developers do not normally estimate size by assessing the number of operators and operands as software science would suggest.

Conclusion

The ABC metric overcomes many of the limitations of the LOC metric. It has a theoretical foundation, is style independent, and has useful mathematical properties. Furthermore, the ABC counting rules for C, C++, and Java are defined to allow accurate cross-language comparisons, thereby enhancing the applicability of the metric.

Preliminary research suggests that the ABC metric provides program size measurements similar to those given by professional developers.

Because of its consistency, the ABC metric should provide better estimates of project development time and program reliability. Specifically, it should provide more accurate comparisons of projects created by different individuals and businesses.

Although the ABC is a promising metric, further research is needed to assess its viability for larger programs. The metric must be compared to LOC and other measures to determine the extent of its benefits. The relationship between the ABC components should be explored further. Finally, the hypothesis regarding size and cohesion should be confirmed and the relationship between the two defined.

Because ABC counting is too tedious and error-prone to perform by hand, I am developing programs for Windows 95/NT that will automate the process. When completed, you'll be able to download them from the website http://www.redmtn.com [26 Jun 2002: these utilities are no longer available]. I encourage you to try ABC instead of LOC. I'd be interested in your results, ideas, and comments.

References

1. Halstead, M.H. *Elements of Software Science*, Elsevier North Holland 1977.
2. Fenton, N. *Software Metrics*, Chapman & Hall 1991.

3. DeMarco, T. *Controlling Software Projects*, Prentice-Hall 1982.
4. Sheppard, M. *Software Engineering Metrics - Volume I*, McGraw-Hill 1993.
5. Kitchenham, B. et al, *IEEE Transactions on Software Engineering*, "Towards a Framework for Software Measurement Validation" (21)12, Dec 1995.
6. Currently unpublished research conducted by the author during December 1996. Details are expected to be published later this year after additional studies have been completed.

Listing 1 – C Function ABC = <5,6,3> [8.4]

```
          static ushort CalculatePageSize(void)
          {
              const ushort MAX_NVRAM_PAGE_SIZE = 256;
              utiny save;
              ushort i, n;

b             outportb(0x800,1);

aca           for(i=1; i < MAX_NVRAM_PAGE_SIZE; i*=2)
              {
a                 n = i + 5;
ab                save = inportb(0x800+n);
b                 outNVRAM(0x800+n,0x55);

bc                if (inportb(0x800+n) != 0x55)
                  {
b                     outNVRAM(0x800+n,save);
                      break;
                  }

b                 outNVRAM(0x800+n,save);
              }

c             if (i == 1)
a                 i = 0;

              return i;
          }
```

Listing 2 – More Complex C Function ABC = <7,12,14> [19.7]

```
       void UpdateSchedule(ADI_EVENT *evt)
       {
           EVENT tempevt;
a          char am=0;

bc         if (!Lock(evt->file, evt->data.rec)
           {
b              DisplayMessage(errorMessage[2], 0x71);
               return;
           }

b          ReadLog(&tempevt, evt->file, evt->data.rec);

bc         if (memcp(&tempevt, &evt->data, sizeof(tempevt)))
           {
bc             if (memcmp(tempevt.src, evt->data.src, sizeof(tempevt.src)) == 0 &&
bc                 memcmp(tempevt.house, evt->data.house, sizeof(tempevt.house)) == 0)
               {
b                  memcpy(&evt->data, &tempevt, sizeof(evt->data));
               }
c              else
               {
b                  UnLock(evt->file, evt->data.rec);
b                  DisplayMessage(errorMessage[5], 0x71);
a                  updateFlag = 1;
               }
           }

aa         evt->data.adistate[0] = evt->adiStatus++;

           switch(evt->adiStatus)
           {
c          case 0x11:
b              memcpy(evt->data.src, device.utility, sizeof(evt->data.src));
               break;

c          case 0x23:
cc             if (device.mode == 19 || device.channel == 2)
b                  memcpy(evt->data.src, device.utility, sizeof(evt->data.src));
               break;

c          default:
c              if (device.mode == 1)
               {
b                  memcpy(evt->data.src, device.utility, sizeof(evt->data.src));
a                  evt->data.adistate[0] = 0;
               }
               break;
           }

a          am = evt->data.amarc[0];
accc       evt->data.amarc[0] = (am >= 0 && am <= 0xF) ? '' : 'C';
       }
```

Listing 3 – C++ Class Method ABC = <5,13,1> [14.0]

```
          static ushort UartStream::Transmit(unsigned char *data)
          {
aa            unsigned char i(0), n = data[1] + 3;

ab            data[n-1] = Checksum(n-1, data);
b             memcpy(itsLastMessage, data, n);
b             WaitReceiveFifoEmpty();

b             disable();
b             SetParityEven();

b             SendByte(data[0]);
b             SendByte(data[1]);
b             SendByte(data[2]);

b             WaitTransmitRegisterEmpty();

b             SetParityOdd();
b             enable();

aca           for(i=3; i < n; i++)
b                 SendByte(data[i]);

b             delay(itsXmtDelay);
          }
```

Listing 4 – Java Class ABC = <13,15,11> [22.7]

```
          public class dump {
              public static void main(String av[]) {
                  final short FIELD_WIDTH = 16;
                  final short HEXADECIMAL = 16;

                  FileInputStream infile;
ab                byte data[] = new byte[FIELD_WIDTH];
a                 int total = 0;

c                 try
                  {
                      int actual;

ab                    infile = new FileInputStream(av[0]);

bc                    while(infile.available() > 0)
                      {
a                         String ascii = "";
                          int i;

ab                        actual = infile.read(data);
a                         total += actual;

aca                       for(i=0; i < actual; i++)
                          {
                              String s;
                              int tmp;

a                             tmp = (int)data[i] & 0xff;
abb                           s = Integer.toHexString(tmp).toUpperCase();

cc                            if (data[i] > 31 && data[i] < 128)
a                                 ascii += (char)data[i];
c                             else
a                                 ascii += ".";

c                             if (tmp < HEXADECIMAL)
b                                 System.out.print("0");

b                             System.out.print(s + " ");

c                             if (i == ((FIELD_WIDTH/2)-1))
b                                 System.out.print(" ");
                          }

c                     if (i < FIELD_WIDTH/2)
b                         System.out.print(" ");

ca                    for(; i < FIELD_WIDTH; i++)
b                         System.out.print(" ");

b                     System.out.print(" " + ascii);
b                     System.out.println();
                      }
                  }
c                 catch(IOException ioe)
                  {
bb                    System.out.println(ioe.toString());
                  }
              }
          }
```


Originally published in C++ Report, June 1997.