5

5 C++ development environment

5.1 INTEGRATED DEVELOPMENT ENVIRONMENT

Usually it is necessary to learn how to use:

- a "command language" for a computer system, (e.g. Unix shell or DOS commands)
- an editor (specialised word processor)
- a "make" system that organizes the compilation of groups of files
- a compiler
- a linking-loader and its associated libraries.

Fortunately, if you are working on a personal computer (Macintosh or PC) you will usually be able to use an "Integrated Development Environment" (IDE).

An IDE packages all the components noted above and makes them available through some simple to use visual interface.

On a Intel 486-, or Pentium- based PC system, you will be using either Borland's C++ environment or Microsoft's C++. On a Macintosh system, you will probably be using Symantec C++. These systems are fairly similar.

Project

They employ a variety of different windows on the screen to present information about a program (usually termed a "project") that you are developing. At least one of these windows will be an editing window where you can change the text of the source code of the program. Another window will display some kind of summary that specifies which files (and, possibly, libraries) are used to form a program. Figure 5.1 illustrates the arrangement with Symantec 8 for the Power PC. The editing window is on the left, the project window is to the right.

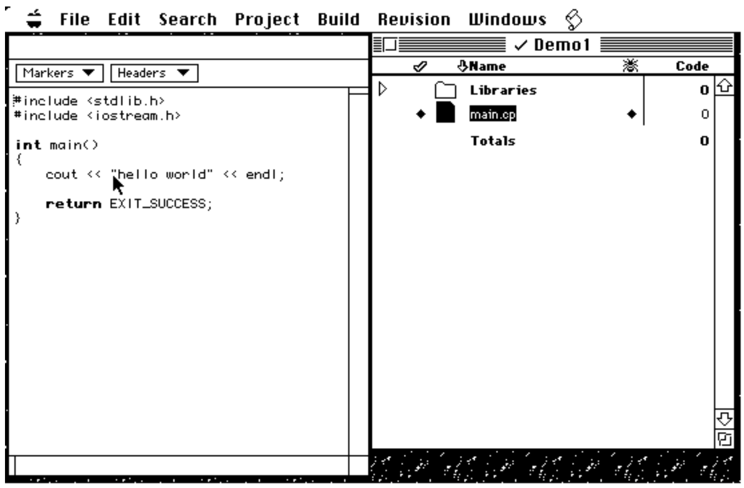

Figure 5.1 Illustration of typical editing and project windows of an example integrated development environment.

The figure illustrates the situation when the Symantec system has been asked to create a new project with the options that it be "ANSI C++ with iostreams". Such options identify things such as which input output library is to be used. The environment generates a tiny fragment of C++ code to get you started; this fragment is shown in the editing window:

```
#include <stdlib.h>
#include <iostream.h>

int main()
{
    cout << "hello world" << endl;

    return EXIT_SUCCESS;
}
```

The two #include lines inform the compiler about the standard libraries that this program is to use. The code form "int main" to the final "}" is a skeletal main program. (It is a real working program; if you compiled and ran it then you would get the message "hello world" displayed.)

The project window lists the components of the program. Here, there is a folder containing libraries and a single program file "main.cp".

The Borland environment would generate something very similar. One of the few noticeable differences would be that instead of being separate windows on the desktop, the project and editing windows would both be enclosed as subwindows of a single large window.

Menu commands

Like spreadsheets, paint programs, and word processor programs, most of the operations of an IDE are controlled through menu commands. There are a number of separate menus:

- File: things like saving to file, opening another file, printing etc.
- Edit: cut-copy-paste editing commands etc.
- Search: finding words, go to line etc
- Project: things like "Run" (run my program!)
- Source: options like "check" and "compile"

The various environments have similar menu options though these are named and organized slightly differently.

5.2 C++ INPUT AND OUTPUT

Because any interesting program is going to have to have at least some output (and usually some input), you have to learn a little about the input and output facilities before you can do anything.

Input/output capabilities defined by libraries

In some languages (FORTRAN, Pascal, etc), the input and output routines are fixed in the language definition. C and C++ are more flexible. These languages assume only that the operating system can provide some primitive 'read' and 'write' functions that may be used to get bytes of data into or out from a program. More useful sets of i/o routines are then provided by libraries. Routines in these libraries will call the 'read' and 'write' routines but they will do a lot of additional work (e.g. converting sequences of digits into the appropriate bit pattern to represent a number).

stdio and iostream libraries

Most C programs use an i/o library called `stdio` (for standard i/o). This library can be used in C++; but more often, C++ programs make use of an alternative library called `iostream` (i/o stream library).

Stream objects

The i/o stream library (`iostream`) makes use of some of the "object oriented" features of C++. It uses "stream objects" to handle i/o.

Now in general an object is something that "*owns a resource and provides services related to that resource*". A "stream object" owns a "stream", either an output stream or an input stream. An output stream is something that takes character data and gets them to an output device. An input stream gets data bytes from some input device and routes them to a program. (A good way of thinking of them is as kinds of software analogues of hardware peripheral device controllers.)

ostream objects

An `ostream` object owns an output stream. The "services" an `ostream` object provides include the following:

- an `ostream` object will take data from an integer variable (i.e. a bit pattern!) and convert it to digits (and ± sign if needed) and send these through its output stream to an output device;
- similarly, an `ostream` object can take data from a "real number" variable and convert to a character sequence with sign character, digits, and decimal point (e.g. 3.142, -999.9, or it may use scientific notation and produce a character sequence like 1.70245E+43);

- an `ostream` object can take a message string (some character data) and copy these characters to the output stream;
- an `ostream` object can accept directions saying how many digits to show for a real number, how much space on a line to use, etc.

istream objects

An `istream` object owns an input stream. The "services" an `istream` object provides include the following ...

- an `istream` object can be asked to find a value for an integer variable, it will check the input stream, miss out any blank space, read in a series of digits and work out the value of the number which it will then put into the variable specified;
- similarly, an `istream` object can be asked to get a "real number" variable, in this case it will look for input patterns like 2.5, -7.9, or 0.6E-20 and sort them out to get the value for the number;
- an `istream` object can be asked to read in a single character or a complete multicharacter message.

Error handling with input and output

It is unusual for an `ostream` object not to be able to deal with a request from a program. Something serious would have had to have gone wrong (e.g. the output is supposed to be being saved on a floppy disk and the disk is full). It is fairly common for an `istream` object to encounter problems. For example, the program might have asked for an integer value to be input, but when the `istream` object looks in the input stream it might not find any digits, instead finding some silly input entered by the user (e.g. "Hello program").

Response to bad input data

An `istream` object can't convert "Hello program" into an integer value! So, it doesn't try. It leaves the bad data waiting in the input stream. It puts 0 (zero) in the variable for which it was asked to find a value. It records the fact that it failed to achieve the last thing it was asked to do. A program can ask a stream object whether it succeeded or failed.

"end of file" condition

Another problem that an istream object might encounter is "end of file". Your program might be reading data from a file, and have some loop that says something like "`istream` object get the next integer from the file". But there may not be any more data in the file! In this case, the `istream` object would again put the value 0 in the variable, and record that it had failed because of end-of-file. A program can ask an `istream` object whether it has reached the end of its input file.

Simple use of streams

The first few programming exercises that you do will involve the simplest requests to iostream objects … "*Hey input stream object, read an integer for me.*", "*Hey output stream object, print this real number.*" The other service requests ("*Did that i/o operation work?*", "*Are we at the end of the input file?*", "*Set the number precision for printing reals to 3 digits*", ...) will be introduced as needed.

Standard streams

A program can use quite a large number of stream objects (the operating system may set a limit on the number of streams used simultaneously, 10, 16, 200 etc depends on the system). Streams can be attached to files so that you can read from a data file and write to an output file.

The `iostream` library sets up three standard streams:

- `cin` standard input stream (reads data typed at keyboard)

- `cout` prints results in an "output window"

- `cerr` prints error messages in an "output window" (often the same window as used by cout); in principle, this allows the programmer to create error reports that are separate from the main output.

You don't have to 'declare' these streams, if your program says it wants to use the `iostream` library then the necessary declarations get included. You can create `iostream` objects that are attached to data files. Initially though, you'll be using just the standard `iostream` objects: `cout` for output and `cin` for input.

Output via cout

Requests for output to `cout` look like the following:

```
int    aCounter;
double aVal;
char Message[] = "The results are:  ";
...
cout << Message;
cout << aCounter;
cout << " and ";
cout << aVal;
...
```

Explanation of code fragment

The code fragments starts with the declarations of some variables, just so we have something with values to output.:

```
int    aCounter;
double aVal;
char Message[] = "The results are:  ";
```

The declaration `int aCounter;` defines `aCounter` as a variable that will hold an integer value. (The guys who invented C didn't like typing, so they made up abbreviations for everything, `int` is an abbreviation for `integer`.) The declaration `double aVal;` specifies that there is to be a variable called `aVal` that will hold a double precision real number. (C and C++ have two representations for real numbers – `float` and `double`. `double` allows for greater accuracy.) The declaration "`char Message ...`" is a bit more complex; all that is really happening here is that the name `Message` is being associated with the text `The results...`.

In this code fragment, and other similar fragments, an ellipsis (...) is used to indicate that some code has been omitted. Just assume that there are some lines of code here that calculate the values that are to go in `aVal` etc.

```
cout << Message;
cout << aCounter;
cout << " and ";
cout << aVal;
```

These are the requests to `cout` asking it to output some values. The general form of a request is:

```
cout << some value
```

(which you read as "`cout` *takes from* some value"). The first request is essential "*Please `cout` print the text known by the name `Message`.*" Similarly, the second request is saying: "*Please `cout` print the integer value held in variable `aCounter`.*" The third request, `cout << " and "` simply requests output of the given text. (Text strings for output don't all have to be defined and given names as was done with `Message`. Instead, text messages can simply appear in requests to `cout` like this.) The final request is: "Please *`cout` print the double value held in variable `aVal`.*"

Abbreviations

C/C++ programmers don't like typing much, so naturally they prefer abbreviated forms. Those output statements could have been concatenated together. Instead of

```
cout << Message;
cout << aCounter;
cout << " and ";
cout << aVal;
```

one can have

```
cout << Message << aCounter << " and " << aVal;
```

Basic formatting of output lines

`cout` will keep appending output to the same output line until you tell it to start a new line. How do you tell it to start a new line?

Well, this is a bit system dependent. Basically, you have to include one or more special characters in the output stream; but these characters differ depending on whether your program is running on Unix, or on DOS, or on ... To avoid such problems, the `iostream` library defines a 'thing' that knows what system is being used and which sends the appropriate character sequence (don't bother about what this 'thing' really is, that is something that only the guys who wrote the `iostream` library need to know).

endl

This 'thing' is called "`endl`" and you can include it in your requests to `cout`. So, if you had wanted the message text on one line, and the two numbers on the next line (and nothing else appended to that second line) you could have used the following request to `cout`:

```
cout << Message  << endl
       << aCounter   << " and  " << aVal << endl;
```

(You can split a statement over more than one line; but be careful as it is easy to make mistakes when you do such things. Use indentation to try to make it clear that it is a single statement. Statements end with semicolon characters.)

Input from cin

Input is similar in style to output except that it uses ">>" (the 'get from operator') with `cin` instead of "<<" (the 'put operator') and `cout`.

If you want to read two integers from `cin`, you can write

```
int aNum1, aNum2;
...
```

```
cin >> aNum1;
cin >> aNum2;
```

or

```
cin >> aNum1 >> aNum2;
```

In general you have

```
cin >> some variable
```

which is read as "`cin` *gives to* some variable".

5.3 A SIMPLE EXAMPLE PROGRAM IN C++

This example is intended to illustrate the processes involved in entering a program, getting it checked, correcting typing errors, compiling the program and running it. The program is very simple; it reads two integer values and prints the quotient and remainder. The coding will be treated pretty informally, we won't bother with all the rules that specify what names are allowed for variables, where variables get defined, what forms of punctuation are needed in statements. These issues are explored after this fairly informal example.

5.3.1 Design

You never start coding until you have completed a design for your program!

But there is not much to design here.

What data will we need?

Seems like four integer variables, two for data given as input, one for quotient and the other for a remainder.

Remember there are different kinds of integers, short integers and long integers. We had better use long integers so permitting the user to have numbers up to ±2000Million.

Program organization?

Main program could print a prompt message, read the data, do the calculations and print the results – a nice simple sequence of instructions.

"Pseudo-code" outline

So the program will be something like:

```
define the four integer variables;
get cout to print a prompt, e.g. "Enter the data values"

get cin to read the two input values

calculate the quotient, using the '/' divide operator
calculate the remainder, using the '%' operator

get cout to print the two results
```

This will be our "main" program. We have to have a routine called `main` and since this is the only routine we've got to write it had better be called `main`.

Here the operations of the routine could be expressed using English sentences. There are more elaborate notations and "pseudo-codes" that may get used to specify how a routine works. English phrases or sentences, like those just used, will suffice for simple programs. You should always complete an outline in some form of pseudo code before you start implementation.

Test data

If you intend to develop a program, you must also create a set of test data. You may have to devise several sets of test data so that you can check all options in a program that has lots of conditional selection statements. In more sophisticated development environments like Unix, there are special software tools that you can use to verify that you have tested all parts of your code. In an example like this, you would need simple test data where you can do the calculations and so check the numerical results of the program. For example, you know that seven divided by three gives a quotient of two and a remainder of one; the data set 7 and 3 would suffice for a simple first test of the program.

5.3.2 Implementation

#including headers for libraries

Our program is going to need to use the `iostream` library. This better be stated in the file that contains our main program (so allowing the compiler to read the "header" file for the iostream library and then, subsequently, check that our code using the iostream facilities correctly).

The file containing the program had better start with the line

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

This line may already be in the "main.cp" file provided by the IDE.

main() routine

In some environments (e.g. Unix) the `main()` routine of a C/C++ program is expected to be a function that returns an integer value. This is because these environments allow "scripting". "Scripts" are "programs" written in the system's job control language that do things like specifying that one program is to run, then the output files it generated are to be input to a second program with the output of this second program being used as input to a third. These scripts need to know if something has gone wrong (if the first program didn't produce any output there isn't much point starting the second and third programs).

Integer value returned by main()

The integer result from a program is therefore usually used as an error indicator, a zero result means no problems, a non-zero result identifies the type of problem that stopped the program.

So, usually you will see a program's `main()` routine having a definition something like:

```
int main()
{
    ...      definitions of variables
    ...      code statements
    return 0; return statement specifying 'result' for
script
}
```

The code "`int main()`" identifies `main` as being the name of a function (indicated by the `()` parentheses), which will compute an integer value. The `main` function should then end with a "`return`" statement that specifies the value that has been computed and is to be returned to the caller. (Rather than "`return 0;`", the code may be "`return EXIT_SUCCESS;`". The name `EXIT_SUCCESS` will have been defined in one of the header files, e.g. `stdlib`'s header; its definition will specify its value as 0. This style with a named return value is felt to be slightly clearer than the bare "`return 0;`").

The integrated development environments, like Symantec or Borland., don't use that kind of scripting mechanism to chain programs together. So, there is no need for the `main()` routine to return any result. Instead of having the main routine defined as returning an integer value, it may be specified that the result part of the routine is empty or "`void`".

In these IDE environments, the definition for a `main()` routine may be something like:

```
void main()
{
    ...   definitions of variables
    ...   code statements
}
```

void

The keyword `void` is used in C/C++ to identify a procedure (a routine that, unlike a function, does not return any value or, if you think like a language designer, has a return value that is empty or void).

Generally, you don't have to write the outline of the `main` routine, it will be provided by the IDE which will put in a few `#include` lines for standard libraries and define either "`int main()`" or "`void main()`". The code outline for `main()` provided by Symantec 8 was illustrated earlier in section 5.1.

Variable definitions

The four (long) integer values (two inputs and two results) are only needed in the main routine, so that is where they should be defined. There is no need to make them "global" (i.e. "shareable") data.

The definitions would be put at the start of the main routine:

```
int main()
{
    long aNum1;
    long aNum2;
    long aQuotient;
    long aRemainder;
    …
```

Naturally, because C/C++ programmers, don't like unnecessary typing, there is an abbreviated form for those definitions ...

```
int main()
{
    long aNum1, aNum2, aQuotient, aRemainder;
    …
```

Choice of variable names

The C/C++ languages don't have any particular naming conventions. But, you will find it useful to adopt some conventions. The use of consistent naming conventions won't make much difference until you get to write larger programs in years 2 and 3 of your course, but you'll never be able to sustain a convention unless you start with it.

Some suggestions:

- local variables and arguments for routines should have names like `aNum`, or `theQuotient`.
- shared ("global") variables should have names that start with 'g', e.g. `gZeroPt`.
- functions should have names that summarize what they do, use multiple words (run together or separated by underscore characters _)

There will be further suggestions later.

Variable "definitions" and "declarations"

This is a subtle point of terminology, reasonably safe to ignore for now! But, if you want to know, there is a difference between variable "definition" and "declaration". Basically, in C++ a *declaration* states that a variable or function exists, and will be defined somewhere (but not necessarily in the file where the declaration appears). A *definition* of a function is its code; a definition of a variable identifies what storage it will be allocated (and may specify an initial value). Those were definitions because they also specified implicitly the storage that would be used for the variables (they were to be 'automatic' variables that would be stored in main's stack frame).

Sketch of code

The code consists mainly of i/o operations:

```
int main()
{
    long aNum1, aNum2, aQuotient, aRemainder;
    cout << "Enter two numbers" << endl;
    cin >> aNum1 >> aNum2;

    aQuotient = aNum1 / aNum2;
    aRemainder = aNum1 % aNum2;

    cout << "The quotient of " << aNum1 << " and "
           << aNum2 << " is " << aQuotient << endl;
     cout << "and the remainder is " << aRemainder << endl;
    return EXIT_SUCCESS;
}
```

Explanation of code fragment

```
int main()
{
    …
}
```

The definition of the `main()` routine starts with `int main` and ends with the final "`}`". The `{ }` brackets are delimiters for a block of code (they are equivalent to Pascal's `begin` and `end` keywords).

```
    long aNum1, aNum2, aQuotient, aRemainder;
```

Here the variables are defined; we want four long integers (long so that we can deal with numbers in the range ±2000million).

```
    cout << "Enter two numbers" << endl;
```

This is the first request to the `cout` object, asking it to arrange the display of the prompt message.

```
    cin >> aNum1 >> aNum2;
```

This is a request to the `cin` object, asking it to read values for two long integers. (Note, we don't ask `cin` whether it succeeded, so if we get given bad data we'll just continue regardless.)

```
    aQuotient = aNum1 / aNum2;
    aRemainder = aNum1 % aNum2;
```

These are the statements that do the calculations. In C/C++ the "/ operator" calculates the quotient, the "% operator" calculates the remainder for an integer division.

```
cout << "The quotient of " << aNum1 << " and "
       << aNum2 << " is " << aQuotient << endl;
 cout << "and the remainder is " << aRemainder << endl;
```

These are the requests to `cout` asking it to get the results printed.

```
return EXIT_SUCCESS;
```

This returns the success status value to the environment (normally ignored in an IDE).

You should always complete a sketch of the code of an assignment before you come to the laboratory and start working on the computer!

Code entry and checking

IDEs use font styles and colours to distinguish elements in the code

The code can be typed into an editor window of the IDE. The process will be fairly similar to text entry to a word processor. The IDE may change font style and or colour automatically as the text is typed. These colour/style changes are meant to highlight different program components. So, language keywords ("return, int, long, ...") may get displayed in bold; #include statements and quoted strings may appear in a different colour from normal text. (These style changes help you avoid errors. It is quite easy to forget to do something like put a closing " sign after a text string; such an error would usually cause numerous confusing error messages when you attempted to run the compiler. But such mistakes are obvious when the IDE uses visual indicators to distinguish program text, comments, strings etc.)

Save often!

You will learn by bitter experience that your computer will "crash" if you type in lots of code without saving! Use the File/Save menu option to save your work at regular intervals when you are entering text.

Compile and "Syntax check" menu options

When you have entered the code of your function you should select the "Compile" option from the appropriate menu (the compile option will appear in different menus of the different IDEs). The IDE may offer an alternative "Syntax check" option. This check option runs just the first part of the compiler and does not go on to generate binary code. It is slightly quicker to use "Syntax check" when finding errors in code that you have just entered.

#include files add to cost of compilations

You will be surprised when you compile your first twenty line program. The compiler will report that it has processed something between 1000 and 1500 lines of code. All those extra lines are the `#included` header files!

Compilation errors

The compiler will identify all places where your code violates the "syntax rules" of the language. For example, the program was entered with the calculation steps given as:

```
aQuotient = aNum1 / aNum2
```

```
    aRemainder = aNum1 % aNum2;
```

(The code is incorrect. The statement setting a value in `aQuotient` doesn't end in a semicolon. The compiler assumes that the statement is meant to continue on the next line. But then it finds that it is dealing with two variable names, `aNum2` and `aRemainder`, without any operator between them. Such a construct is illegal.) When this code was compiled, the Symantec compiler generated the report

```
File "main.cp"; Line 11
Error:   ';' expected
```

Error messages are commands that select source code lines with errors

The IDE environments all use the same approach when reporting the compilation errors. Each error report actually consists of commands that the IDE can itself interpret. Thus, the phrase File "main.cp"; can be interpreted by Symantec's IDE as a command to open an editing window and display the text of the file main.cp (if the file is already displayed in a window, then that window is brought in front of all other windows). The second part of the error report, Line 11, is a directive that the IDE can interpret to move the editing position to the appropriate line. The rest of the error report will be some form of comment explaining the nature of the error.

If you do get compilation errors, you can select each error report in turn (usually, by double clicking the mouse button in the text of the error message). Then you can correct the mistake (quite often the mistake is on a line just before that identified in the report).

Generally, you should try to correct as many errors as possible before recompiling. However, some errors confuse the compiler. An initial error message may be followed by many other meaningless error messages. Fixing up the error identified in the first message of such a group will also fix all the subsequent errors.

Building a project

If your code compiles successfully you can go ahead and "Build" your project. The build process essentially performs the link-loading step that gets the code of functions from the libraries and adds this code to your own code.

Link errors

During the build process, you may get a "link error". A link error occurs when your code specifies that you want to use some function that the IDE can not find in the libraries that you have listed. Such errors most often result from a typing error where you've given the wrong function name, or from your having forgotten to specify one of the libraries that you need.

Running the program

You can simply "Run" your program or you can run it under control from the IDE's debugger. If you just run the program, it will open a new window for input and output. You enter your data and get your results. When the program terminates it will display some message (e.g. via an "Alert" dialog); when you clear ("OK") the dialog you are returned to the main part of the development environment with its editor and project windows.

Debugger

Running under the control of a debugger takes a little more setting up. Eventually, the IDE will open a window showing the code of your main program with an arrow marker at the first statement. Another "Data" window will also have been opened. You can then control execution of your program, making it proceed one statement at a time or letting it advance through a group of statements before stopping at a "breakpoint" that you set (usually by clicking the mouse button on the line where you want your program to stop).

Stack backtrace

Figure 5.2 illustrates the windows displayed by the Symantec 8 debugger. The left pane of the window summarizes the sequence of function calls;. In this case there is no interesting information, the data simply show that the program is executing the main program and that this has been called by one of the startup routines provided by the IDE environment. (In Borland's IDE, this "stack backtrace" information is displayed in a separate window that is only shown if specifically requested by menu command.)

Code of current procedure

The right pane of the window shows the code of the procedure currently being executed. The arrow marks the next statement to be executed. Statements where the program is to stop and let the debugger regain control are highlighted in some way (Symantec uses diamond shaped check marks in the left margin, Borland uses colour coding of the lines). If you execute the program using the "Step" menu command you only execute one statement, a "Go" command will run to the next breakpoint.

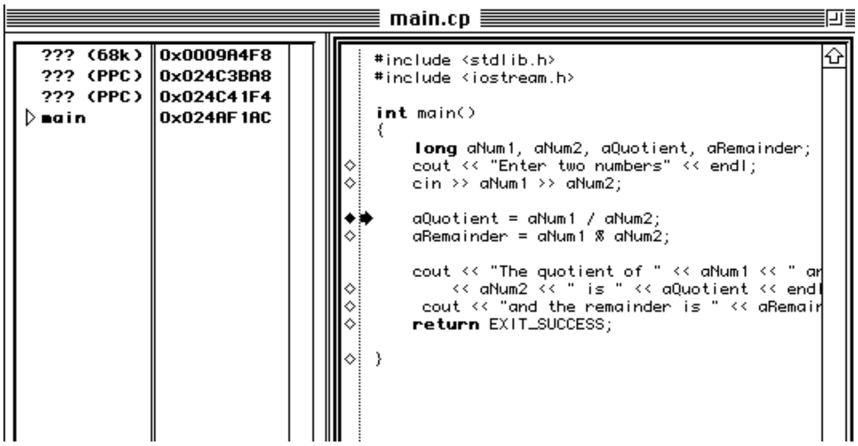

Figure 5.2 Display from a debugger.

The contents of variables can be inspected when the debugger is in control. Typically, you select the variable with the mouse, its value will be displayed in the separate Data window (it may be necessary to use a "Data" or "Inspect" menu command).

You should learn to use the debugger starting with your very first programming exercises. Initially, using the debugger to step through a program one statement at a time helps you understand how programs get executed. Later, you will find the debugger very useful for eliminating programming errors. It is easy to have programming errors such as defining a collection (array) with four data elements and then coding a loop slightly wrongly so that the program ends up checking for a fifth data element. Such code would be syntactically correct and so would be accepted by the compiler, but would either "bomb out" when run or would generate an incorrect result. It is very hard to find such an error if all you know is that the program "bombs". It is relatively easy to find such errors if you are able to use the debugger to stop the program and check what is going.