



The University of Sydney

**HOW BEST TO TACKLE AND TEACH
SYNTAX ERROR CORRECTION**

TECHNICAL REPORT NUMBER 529

JULY 2001

Sarah K. Kummerfeld and Judy Kay

ISBN 1 86487 385 X

**Basser Department of Computer Science
University of Sydney NSW 2006**

How Best to Tackle and Teach Syntax Error Correction

Sarah K. Kummerfeld

Department of Computer Science
University of Sydney, 2006, Australia
sarah@cs.usyd.edu.au

Abstract

Syntax error correction is an important part of the debugging process. Yet there has been little research investigating how programmers approach syntax errors correction and how to teach beginner programmers to fix errors efficiently. This study investigates how beginners and experts approach syntax error correction in C/C++. Qualitative data was collected by observing programmers with varying levels of experience as they corrected syntax errors. Half of the participants were given access to a web-based compile-error reference guide. This explains each error message and gives concrete example code showing possible causes and corrections. The study indicates that general and language specific programming experience provide strategic skill for correcting errors and greater depth of understanding of the error messages themselves. However, beginners can be almost as efficient as more expert users with access to reference material that explains the more unintuitive compile error messages.

Keywords: debugging, syntax error, teaching and learning programming, strategic debugging skills, expert/novice study

Introduction

Debugging pedagogy has been an active area of research (Joni, Soloway, Goldman and Ehrlich 1983, Spohrer, Soloway and Pope 1985, Johnson, Soloway 1985). There has been much work done to develop methods and tools to help students perform efficient testing (Martin 1996). However, this work has been focused almost exclusively on runtime and logical errors rather than compile-time errors (Yazdani 1986, Ohlsson 1987).

There are virtually no resources to assist with syntax error correction. The few studies that have been conducted focused on enumeration of different classes of syntax errors (Draper 2000 and references therein).

Anecdotal experiences indicate that students using an unfamiliar or new programming language waste considerable time correcting syntax errors. Studies have shown that excessive time spent on correcting syntax problems can be detrimental to long-term success as students become disheartened with programming (McKeown and Farrell 1999, Martin 1996).

Reducing the cognitive load, by easing the burden of syntax error correction, will also enable students to learn more quickly and efficiently.

This phenomenon is particularly noticeable in students learning a language like C for the first time. C is a large language that requires an understanding of a broad range of concepts for even very simple programming tasks.

The students in this study had first learnt Blue (a small-talk like language); using a development environment with very good debugging facilities.

The purpose of the Study

This study investigates syntax error detection and correction using a similar approach to that taken for other types of debugging (e.g. Draper 1996). Three questions were considered:

1. How do programmers (novice and expert) approach syntax error correction?
2. What is the best way to approach syntax error correction?
3. How can syntax error correction best be taught or supported?

Description of the Study

To study how novice and expert programmers approach syntax error correction, a group of students was selected. They had varying degrees of programming experience and familiarity with C/C++ (Figure 1/appendix A).

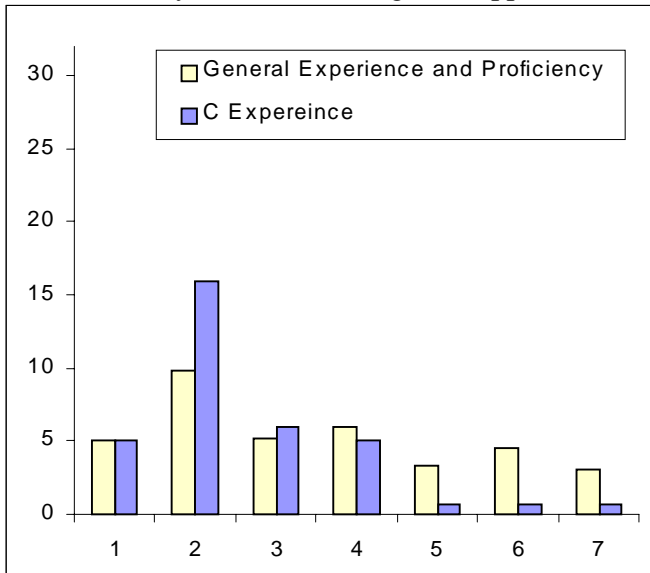


Figure 1: Summary of subject experience. C programming experience was estimated by adding the number of years the subject had been programming in C and the number of major projects they had coded in C. General proficiency includes years of programming experience, university grades, year level and C experience. Subjects 5, 6 and 7 were novice C programmers; they began learning C 8 weeks before the study. Subjects 1, 3 and 4 were more experienced and 2 was more experienced again.

A set of 10 small programs, each with one or two syntax errors was developed. These were designed to include a cross-section of different problems in no particular order. They were chosen because the compile error messages were judged as nonnutritive and so likely to cause difficulties in debugging. Figure 2 provides a summary of the errors.

Monitoring software was developed to record the time of compilation, the error that occurred and the program code at the time. This was written in *Python* and designed to look like a standard shell, with a restricted set of commands (Figure 3).

A web-based reference guide was developed to catalogue some common C/C++ compile errors. The errors were compiled from personal coding experience. Each error message is explained with examples highlighting the problem and at least one possible correction. Figure 4 shows a sample entry from the reference guide. The actual text of the error message is shown at the top of the entry. An explanation is in the top left box of the table. The lower row shows a code example on the left, the error in red. Its correction is to the right in green. For a complete listing of the 16 errors, see appendix C.

On Completion of the exercise, subjects were given a short questionnaire. This asked them about their C and general programming experience, university marks. Those who were given access to the reference were asked how useful they thought it was and whether they would be likely to use such a guide in the future.

Program	Error
A	- Semicolon is missing after class <i>video_obj</i> - <i>vector<*video_obj> catalogue;</i> the * in wrong place
B	- missing <i>#include <stdio.h></i> - missing <i>/*</i> on comment
C	- Unused variable ' <i>char *line</i> ' - The return statement for function, <i>doit</i> is within an if statement (it may not necessary return anything even though it should).
D	- The declaration of function, <i>statify</i> , is missing a return type.
E	- Semicolon missing after class <i>arrive</i> - The variable called <i>type</i> is declared twice, once as an <i>int</i> later as a <i>string</i> .
F	- Declaration of the <i>main</i> function is missing the <i>char *</i> type in its argument list. - A variable called <i>the_char</i> not declared
G	- <i>Node</i> is used as both the name of a struct and the name of a variable.
H	- The function called <i>printReverse</i> is prototyped with the wrong arguments - There is a conditional statement with a single <i>=</i> instead of <i>==</i> .

Figure 3: Summary of syntax errors. Each program (A-H) contained one or two errors. This table summarises the problems with each program.

```

xterm
lsarah@eth Questions$ syntax_error_exercise joe_blogs
make <program name> To compile a program
run <program name> To run the previously compiled program
"," to end session
> make a.cc
a.cc:14: semicolon missing after declaration of `video_obj'
a.cc:15: extraneous `int' ignored
a.cc:15: semicolon missing after declaration of `class video_obj'
a.cc: In function `int main()':
a.cc:15: parse error before `>'
a.cc:22: `catalogue' undeclared (first use this function)
a.cc:22: (Each undeclared identifier is reported only once
a.cc:22: for each function it appears in.)
a.cc:27: confused by earlier errors, bailing out
> █

```

Figure 3: Screen shot of exercise monitoring software. The user starts the program providing their name as the first parameter (used as the file name for the log). A short explanation of the system is provided. The program in file 'a.cc' has been compiled, the error messages are shown.

ANSI C++ forbids declaration 'function' with no type	
The second argument is missing its type.	
function(int a, b)	int function(int a, int b)
{	{
int ans = a*a - b*b;	int ans = a*a - b*b;
ans += 1;	ans += 1;
return ans;	return ans;
}	}

Figure 4: Sample entry from the reference guide. This entry explains the error caused by failure to include the type of a variable in an argument list (see appendix C).

Results and Discussion

The time taken by subjects to solve each problem and the number of times they compiled each program was used as a measure of efficiency. These results are summarised in figure 5. These values been used as a measure of the subject's efficiency at correcting syntax errors. Subject 7 failed to correct 7 of the 8 errors despite spending over an hour on the exercise.

Think-aloud protocols are widely accepted as a technique for qualitative analysis of user-interfaces and learning through studying only a few subjects (Newman and Lamming 1995, Piaget 1974). Think-aloud protocols have been used to determine how programmers approach syntax error correction. The results are described as 9 observations. Appendix B has a full summary of the think-aloud.

1. Experienced programmers used strategies to correct particular syntax errors.

This was error-specific; meaning that different languages or even different compilers require a new set of strategies to be learned. An example of this was:

```

a.cc:14: semicolon missing after declaration of
`video_obj'

```

The more experienced C programmers moved firstly to the line indicated (14) and after looking briefly at this line, they proceeded to look above for the declaration of the video_obj class. This allowed them to fix this error very quickly. In contrast, the less experienced C programmers did not know what to do once they had read line 14.

Another example of error-specific strategy was in response to:

```

h.cc:81: warning: suggest parentheses around
assignment used as truth value

```

The experienced programmers asked themselves whether this was intended to be an assignment or a conditional statement missing its second '='. This response showed that they understood the implications of this error as well as knowing how it could be fixed.

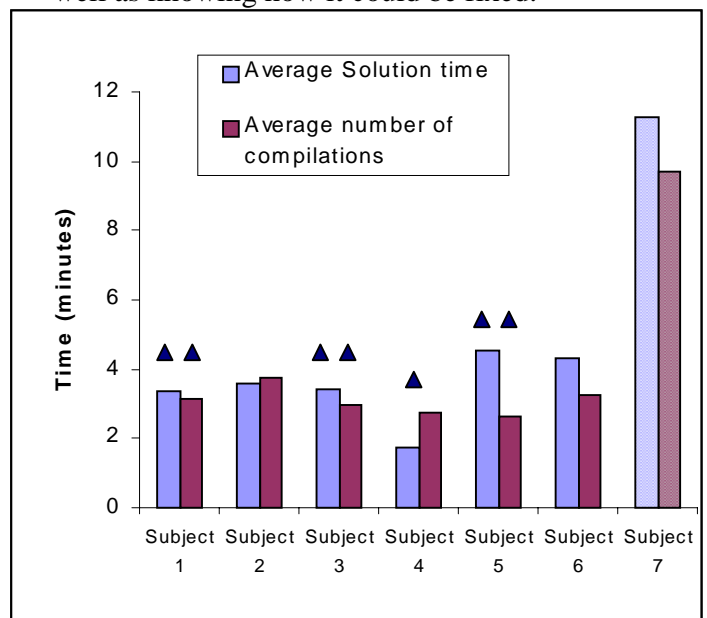


Figure 5: Average time and number of compilations to solution. The subjects who were given access to the reference guide are marked with a ▲. Those who actually looked at the guide are marked with a second ▲. Subject 7 is grayed out to indicate they failed to complete 7 of the 8 exercises. All other subjects solved all problems.

The less experienced programmers simply added parentheses around the assignment statement. This did not affect the time taken to reach the successful compilation stage.

However, if the wrong correction were made, the time to generate a logically correct program would be greater.

2. Experienced programmers used a range of strategies to approach unfamiliar error messages. Some of the compile errors were unfamiliar to even the most experienced programmers. They used a set of generic strategies to find and correct the problem. The program B error due to missing standard IO header file caused a large number of errors, beginning with:

```
b.cc:4: type specified omitted for parameter
b.cc:4: parse error before `*'
b.cc:5: `FILE' was not declared in this scope
b.cc:5: `stream' was not declared in this scope
b.cc:5: variable or field `readFiles' declared
void
```

The less experienced programmers appeared to panic at the large number of errors. They began reading the first error and once they established that they did not know what it meant, they immediately asked for assistance or used the reference guide. The more experienced programmers also began with the first error. However, when this did not shed light on the problem, they began reading the later errors and the code around the lines specified in the error message. After reading a few more lines of the compiler output it was clear that the standard IO library was needed.

3. When strategies failed, both novice and expert programmers tried erratic alterations to code in an attempt to correct syntax errors. This finding is consistent with previous studies which showed that students would ‘tinker’ almost randomly with code as a final attempt at solving syntax errors (Martin 1996 and references therein). Some of the compilation errors proved difficult for the subjects. Once they had tried a logical/strategic approach to fixing the problem, they began making what seemed like erratic changes; compiling the code after each

alteration, often introducing new problems. A striking demonstration of this observation occurred in program H:

```
h.cc:5: too many arguments to function
`printReverse(char *, int)'
h.cc:93: at this point in file
```

The initial response was to remove the third argument from the declaration of printReverse. This caused a linking error:

```
Undefined symbol first referenced
printReverse(char *, int) in file
/var/tmp/cca21sun.o
ld: fatal: Symbol referencing errors. No output
written to exec
collect2: ld returned 1 exit status
```

This seemed to panic three of the subjects; prompting them to try making “apparently” adhoc alterations to the code. These included: deleting the first or second arguments instead of the third and changing the variables’ names in the function prototype!

4. The compile-error reference guide provided enough help for the novices to perform favorably compared to the experts in terms of time efficiency. This was also true for more experienced programmers who encountered an error that they had not seen before or did not understand.

Program A included a variable declaration with a '*' in the wrong location, causing the error message:

```
a.cc:16: parse error before `>'
a.cc:22: `catalogue' undeclared (first use this
function)
a.cc:22: (Each undeclared identifier is
reported only once
a.cc:22: for each function it appears in.)
a.cc:27: confused by earlier errors, bailing
out
```

The experts identified the error as soon as they looked at the problem line. By contrast, less experienced programmers were not sure what the error message meant even after looking at the code. However, after looking at the reference guide they were able to correct the problem immediately. This suggests that a syntax error reference may help beginners correct errors more quickly. Further study would be needed to decide whether the guide helps beginners develop the all-important generic strategies for error correction.

5. The examples were an essential part of the reference guide explanation. The subjects who used the reference guide seemed not to understand the problem until they looked at the example code. This suggests that concrete examples are essential for teaching syntax error correction.

6. The more experienced programmers were less inclined to use the reference guide, even when they did not understand an error message. However, when they did use the guide, they were able to correct problems very quickly. An example of this was the error in program A described above (observation 4).

For over 4 minutes, subject 3 inspected the error message and code before consulting the reference. In contrast, subject 1 read the error and looked briefly at the code (for less than 1 minute) before looking at the reference guide. As a result, the total time taken for this problem by subject 3 was almost twice as long as for Subject 1. These two programmers have a comparable level of programming experience.

7. Error messages are often misleading. The examples above clearly illustrate this point. For example, line numbers indicating the location of the error are frequently wrong.

8. Syntax error correction can be very time consuming. This trial was developed with a set of 10 small programs (28 to 459 lines) each with one or two syntax errors. During testing of the first subject, the number of programs was revised down to 8 to keep the exercise under 45 minutes in duration. On average, the subjects took just over half an hour to correct all 8 programs. Even the experienced programmers occasionally took as long as 8 minutes to correct a single error (without the reference guide). Quantitative comparisons between those who did and did not use the guide are not possible with this sample size. However, the post-trial questionnaire showed that, all of the subjects who used the reference thought it was helpful and said they would be very likely to use such a guide in future if it

were available. One individual asked if the guide developed for this study could be released on the web.

9. Experience is not the only factor that affects syntax error correction efficiency.

The time it took subjects to reach successful compilation may also be related to programming proficiency. The stronger-students (in terms of university results) tended to solve the problems more quickly. This was particularly noticeable for the novice C programmers. Further investigations should take this into account when evaluating syntax error correction speeds. It may be that the weaker students benefit more than the more able students from using a reference guide. This is consistent with previous work into tailoring learning to cater variation in students' ability (Corbett and Anderson 1995).

Conclusion

Syntax error correction is the first step in the debugging process. It is not possible to continue program development until the code compiles. This means it is an important part of the error correction process. Compilers for complex languages like C and C++ often produce unintuitive compile errors (Martin 1996).

In conclusion, we return to the initial questions posed in the introduction:

1. How do programmers (novice and expert) approach syntax error correction?

This study found that experts tended to use error specific and general strategies to correct syntax errors.

The C/C++ experts were able to respond quickly to errors through what appeared to be learned strategies and a deep understanding of error messages.

The experts also used generic strategies to approach unfamiliar errors. Programmers at all levels (without access to reference material) began making what seemed adhoc changes to the code once their language-specific

knowledge and generic error correction strategies had been exhausted.

Beginners tended to go directly to the reference guide when confronted with an error message that they had not seen before or did not understand. In contrast, the more experienced programmers tended to make an attempt at fixing the bug before referring to the guide.

2. What is the best way to approach syntax error correction?

The key to efficient syntax error correction was the use of strategies; specific strategies for familiar errors and generic heuristics for unfamiliar errors. Language-specific experience was important for development of both specific and generic strategies. The more experienced C programmers seemed to have a deeper understanding of the error messages. General programming experience was less important, but did improve the subjects' ability to correct unfamiliar errors using generic strategies.

3. How can syntax error correction best be taught or supported?

A reference could act as a primary aid for learning syntax error correction. While it is not clear whether such a guide would help students learn general debugging strategies, it would save them wasting time while they are developing the error specific skills. Reducing the burden of syntax error correction early is essential for preventing frustration (Martin 1996).

Further development of teaching resources would also be beneficial. In particular, Draper (1996) suggests students should provoke errors in their own code to gain a deep understanding of syntax error messages. For example, students could be asked to introduce an error by removing a semicolon from working code and observe the outcome. In fact, students might be asked to construct their own guide from such experiments. However, it seems students may not be motivated to keep track of such errors (Greening, per com 2000).

One limitation of this study is that it separated the program writing from the error correction

since the author provided the set of test programs. There may be differences in the programmers' approach when they are debugging their own code as opposed to foreign code.

One may argue that compilers should provide better error messages. While this may improve the situation, it would not be realistic to provide detailed explanations or concrete examples. The examples seemed to be critical for error message comprehension.

Clearly, the key to fast syntax error correction is being able to associate an error message with a problem or set of problems. It is unrealistic to expect students to be aware of every compile error. However, techniques or heuristics for approaching errors combined with reference material will lead to efficient syntax error correction.

References

- Corbett, A.T., and Anderson, J.R., 1995, Knowledge Tracing: Modeling the Acquisition of Procedural Knowledge, *User Modeling and User-Adapted Interaction*, 4(4):253-278.
- Draper, S.W., 1996, Programming skills, visual layout design, and unjustifiably useful testing, reports in the psychology of programming (available online at: <http://www.psy.gla.ac.uk/~steve>).
- Draper, S.W., 2000, Discussion on error types, Seminar on classification of errors, available online at: <http://www.psy.gla.ac.uk/~steve/talks>).
- Johnson, W.L., Soloway, E., 1985, PROUST: Knowledge-Based Program Understanding, *IEEE Transactions on Software Engineering*, 11(3):267-275
- Joni, S.N., Soloway, E., Goldman, R., and Ehrlich, K., 1983, Just So Stories: How The Program Got That Bug, *Proceedings of the SIGCUE/SIGCAS Symposium on Computer Literacy*.
- Martin, J.L., 1996, Is Turing a better language for teaching programming than Pascal, Honours Dissertation, University of Stirling, Department of Computer Science (available online at: <http://www.holtsoft.com/turing/essay.html>)
- McKeown J. and Farrell, T., 1999 Why We Need to Develop Success in introductory programming courses, CCSCPC. (Available online at: <http://homepages.dsu.edu/mckeownj/CPCCSCpaper.html>)
- Newman, W.M., and Lamming, M.G., 1995, *Interactive System Design*, Addison-Wesley.
- Ohlsson, S., 1997, Some properties of Intelligent Tutoring Systems, in *Artificial intelligence and education*, eds Lawer, R.W., and Yazadani, M. Ablex publishing.
- Piaget, J., and Inhelder, B., 1974, *The child's construction of quantities*, Routledge and Kegan Paul.
- Spohrer, J.C., Soloway, E., and Pope, E., 1985, A Goal/Plan Analysis of Buggy Pascal Programs, *Human-Computer Interaction*.
- Yazdani, M., 1987, Intelligent tutoring systems and overview, in *Artificial intelligence and education*, eds Lawer, R.W., and Yazadani, M. Ablex publishing

Appendix A: Summary of Subject Experience Questionnaire

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5	Subject 6	Subject 7
Year	3	4	3	3	2	2	2
Years of Programming	2.5	5	3	2.5	4	2	10
Languages	Blue, Java, C++	C, C++, Java, VB, Pascal, Python, ksh, sh, bash, PL/SQL, Motorola, assembly, Prolog, List	C, C++, Java, Blue, Mips, assembly, Python, Prolog	C, C++, Blue, Java, Python, Perl, Shell	Blue, Eiffel, Java, C, C++, mips assembly, Delphi, basic	Blue, Java, Bash, Eiffel, JSP, C++, Python	Basic, Blue, Java, Eiffel, C++
Comp1001 Results	C (Adv, 1998)	n/a	D (1998)	D (1998)	D (Adv 1999)	HD (Adv 1999)	C (Adv 1999)
Comp1002 Results	C (Adv, 1998)	n/a	P (1998)	HD (Adv 1998)	D (Adv 1999)	HD (Adv 1999)	C (Adv 1999)
DDS Results	P (Adv, 1999)	D (1997)	P (1999)	D (Adv 1999)	C (Adv 2000)	D (Adv 2000)	C (2000)
PPU Results	C (1999)	D (1997)	C (1999)	C (Adv 1999)	n/a	n/a	n/a
OOS Results	C (2000)	n/a	P (2000)	n/a	n/a	n/a	n/a
SE Results	n/a	C (1998)	n/a	n/a	n/a	n/a	n/a
C/C++ experience (yrs)	1	4	2	1	0.15	0.15	0.15
Projects in C/C++	4	12	4	4	0.5	0.5	0.5
Did you find the reference helpful?	Yes	n/a	Very helpful	Found there was no need to look at it.	Yes	n/a	n/a
How likely would you be to use such a resource in future programming projects?	Very likely	n/a	Very likely	Very unlikely	n/a	n/a	n/a
C experience	5	16	6	5	0.65	0.65	0.65
General Programming Experience	2.5	4	2.5	2.5	1.5	1.5	1.5
University CS marks	4.5	5.3	4.00	7.5	6	9.2	5
Year level	3	4	3	3	2	2	2
General Experience and Proficiency	5	9.76	5.16	6	3.38	4.45	3.05

Appendix B: Think-aloud and Observations

Subject 1

a.cc

Immediately adds ';' to class video_obj.

Reads the next error, immediately goes to the reference guide, selecting the entry 'parse error before <'. Comments 'Oh! I see, i've never had that before'. Fixes the problem immediately.

b.cc

Shows anxiety at the large number of errors

Asks if FILE is from the iostream library (due lack of familiarity with C). Answer give (since this is a language issue, the subject found the error and knew how to remedy the problem, but was missing the detailed knowledge of C).

c.cc

Comments that one of the errors is just a warning and doesn't matter.

Looks for return statement, adds one to return an int as the end of the function.

d.cc

Goes to line 128. Reads the declaration, goes back up to the class prototype, notices it has a void return type, adds this to the function declaration.

e.cc

Goes to line 70, then reads the error and right away adds ';' after the declaration of class Arrive.

Looks at the declaration of type, changes it to string. A new error occurs, fixes this by removing the second declaration of type, but leaving it as a string.

f.cc

Reads the first error, immediately goes to the reference guide. Reads the section and then asks whether argc should be a char * or an int. Being a question about the language conventions rather than about the understanding of the error, the answer was provided.

g.cc

Reads the error, sees that n is declared as Node * n. Comments, "but it is declared". Checks that class Node is declared. Looks at the reference guide. Begins searching for a variable called Node. Finds it, changes the variable's name.

h.cc

Reads the warning, immediately adds an extra '=', comments 'I have that all the time'. Approaches the second error by trying to delete each argument. Then tries various combinations before going to line 5 and seeing the correct declaration.

Subject 2

a.cc

Reads a number of errors, comments that he will "handle all the semicolons first". Goes to line 14 and comments that the compiler "never gives the right line". Looks above, adding a ';' after video_obj.

Reads the extraneous int error, goes to the line but can't find the error. Again looks at line 16, commenting he is not use to the STL. Notices the *, comments "this probably wont be it, but i'll give it a go" and removes it. The problem persists. Looks at the declaration of main, wonders about the return type. Starts checking futher afield, looks at the class declaration, ontising the lack of explicit constructor. Decides to move on and come back to the problem later.

-- returns

Deletes the *, adding a &. Error remains. Tries * on the other side, asking if that was what there was originally.

b.cc

Noties FILE, check above since it requires stdio.h. Adds #include <stdio.h>. Wonders about the error on line 4 giving this error, begins reading more of the errors before deciding that they all may be caused by the missing inclusion of sdio.h.

c.cc

Goes to the declaratin of line, comments it out. Looks at the declaration of the function, deciding if it should return somehting or change the return type to void. Decides on chaging the return type.

d.cc

Goes to line 128, then reads the error. Check for the simulation class. Checks for its return value. Notices that it should be void. Comments that the second error must be handled by this correction. Goes to line 17 and looks at the other errors.

e.cc

Goes to line 70, looks above for the declaration of Arrive. Reads the "multiple types" errors, commenting this is unfamiliar. Goes to line 78 but can't see what could be wrong. Removes the semicolon at the end of the line. Goes to line 16, comments that this looks fine. Reads the next error Goes to the previous declaration, considers the scope. Changes the variable name.

f.cc

Goes to line 17, notices missing parameter type, adds char. Adds declaration of the _char. Errors persist, tries adding const before parameter char argv. This fails to correct the problem. Looks again at the declaration and makes it char * argv.

g.cc

After lookng at the line 'Node * n', show surprise at n being undeclared. Looks for declaration of Node calss.Asks about any Node.h file. Wonders why there is no error about Node earlier in the program. Looks up to see if n is being declared as a differnet type earlier. Finds previous declaration with Node as variable name. Checks the other errors, comments tha they should now be corrected.

h.cc

Looks at lin 81, comments on bas style of missing parentheses. Asks whether it was intended to be == or = with parentheses. Read shte next two errors together. Tries to make sence of he arguements in terms of the program context. Removes the third arguement. Linking error (undefined function). Begins looking for other declaration of printReverse. Finds it, notices stream as third arguement, puts it back below and changes the forward declaration.

Subject 3

a.cc

Goes to error line, looks above and adds ';' to the class.

Looks at the declaration on line 16, goes to the reference guide, going to the 'parse error before >' entry. Immediately moves the * correct position.

b.cc

Looks at the first few errors, adds #include <stdio.h>, looks again at the errors before again compiling.

c.cc

Comments out the declaration of line, checks for use of line in the function.

Searches for returns, doesn't find any, so adds void.

d.cc

Looks at the error, scrolls through the program looking for the problem (doesn't go directly to the line).

Looks at the reference guide but can't find a comparable error message. Checks for return within the statify function, adds void.

e.cc

Scrolls looking for the Arrive class. Rather than going direct to the line, scrolls through the code. Finds string type and int type, changes int type to int type_int. Searches for type, identifying which variables need to be replaced by type_int. Since there are none, comments out the declaration line.

f.cc

After looking at the declaration for some time, adds char * before argv.

Subject 4

a.cc

looks at line 14, comments "doesn't look like an error"

looks above, finds missing ';' at the end of class videoe_obj. Corrects the mistake.

Tries changing #include <vector> to #include <vector.h>

Notices '*' in wrong spot

b.cc

Sees a huge number of errors, Looks at tutorial, looks back at the parameters, considers FILE

Adds #include <stdio.h>

c.cc

Goes to line 13 and deletes char * line;

Looks at function doit's declaration, comments "meant to return an int"

Searches for return statements

Adds a return at the bottom

d.cc

Looks at the prototype of statify, comments "it should be void"

Scrolls to function statify and adds void

e.cc

Searches from the beginning of the document for class missing a ';'.

Finds the class missing a ';'.

During compilation, begins looking for other errors, finds declaration of type as string.

f.cc

Immediately locates the missing type in the declaratin of main

g.cc

Looks at declaration of n (Node * n)

Searches above, within the function, finds int Node, comments this out.

h.cc

Goes to declaration of printReverse (line 93), deletes the last paramter, responding to the error message 'too many arguements to function'. Linking error occurs, undefined function.

Looks at the prototype of printReserve (line 5), notes 2 parameters. Searches later in the program to decide wheither to add or remove argumeents.

On reading the warning, 'suggested parentheses around assignment used as truth value', immediately goes to the line and adds and extra '=' at to make it '=='.

Subject 5

a.cc

Reads the error, goes to line 14. Re-reads the error. Looks for video_objs class. Cant see it at the line indicated. Goes to the reference guide. Tries adding ';' after main. Reads the next error, goes directly to the reference and then to line 16, fixing the problem immediately.

b.cc

Reads the first error, looks at the line. Goes to the reference guide, looking up undeclared variable error. Compares the examples to the compile error. Notices printf undeclared is one of the error, corresponding to missing inclusion of a library class. Adds #include <stdio.h>

c.cc

Looks the program, then at the error messages. Looks for 'line'. Checks it is not used, then comments it out. Reads the next error 'control reaches end of non-void function'. Comments on being unsure of what that means. Looks at the line indicated. Still unsure, searches for 'control reaches'. Reads the tutorial entry. Searches code for any return statements. Replaces the return type with void.

d.cc

Goes to line 128, then reads the error. Searches the reference guide for 'with no type'. Adds void to the declaration.

e.cc

Reads the error, then goes to the line indicated. Looks for Arrive, notices it is above and missing a ';'.

Reads the next error and goes to the line indicated. Notices that string is declared once as int and once as string. Scrolls to the previous declaration, commenting it.

f.cc

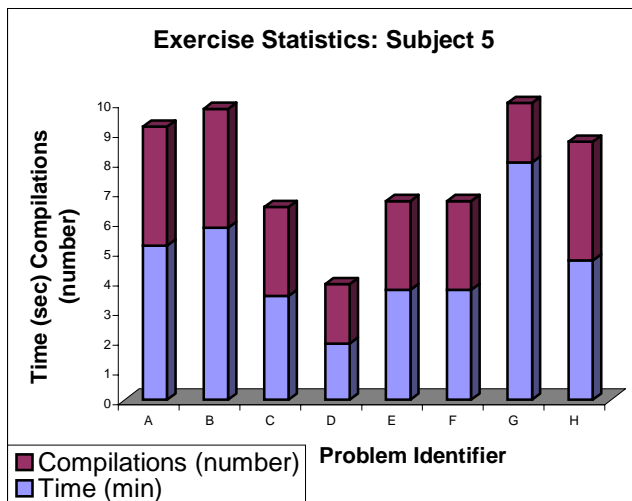
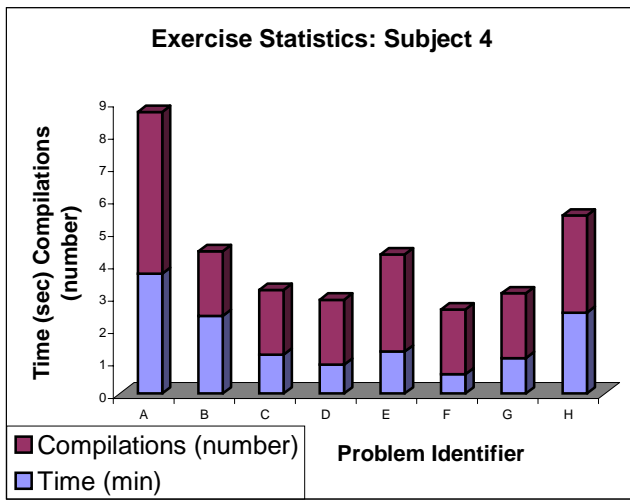
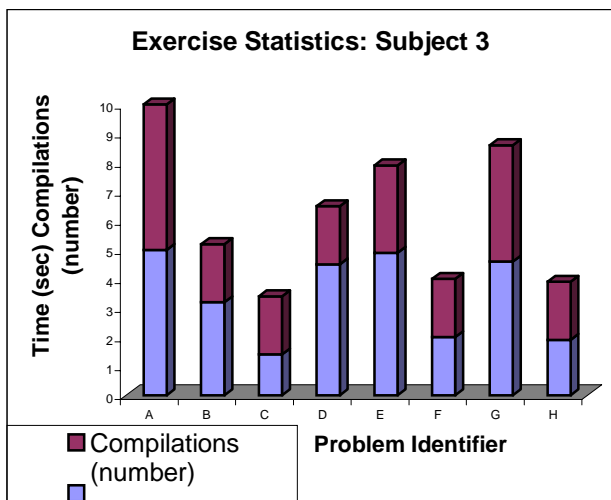
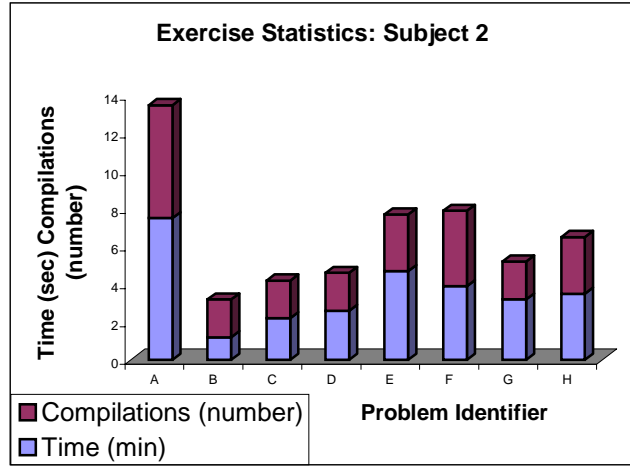
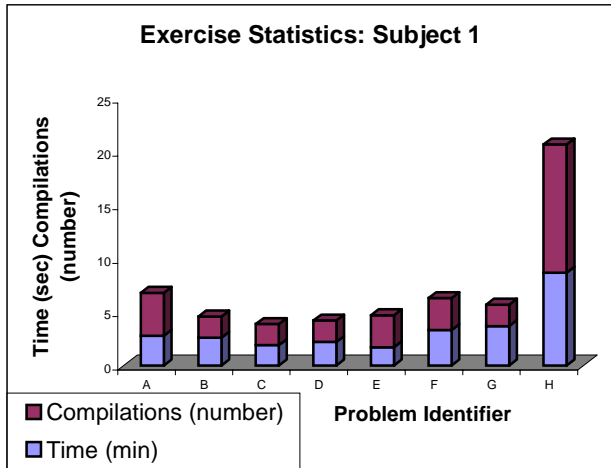
Reads the first error and looks at the code. Cant see a problem. Searches the reference, reads the explanation and fixes the problem. Goes to the line indicated, comments “you just haven’t declared this”, adds it.

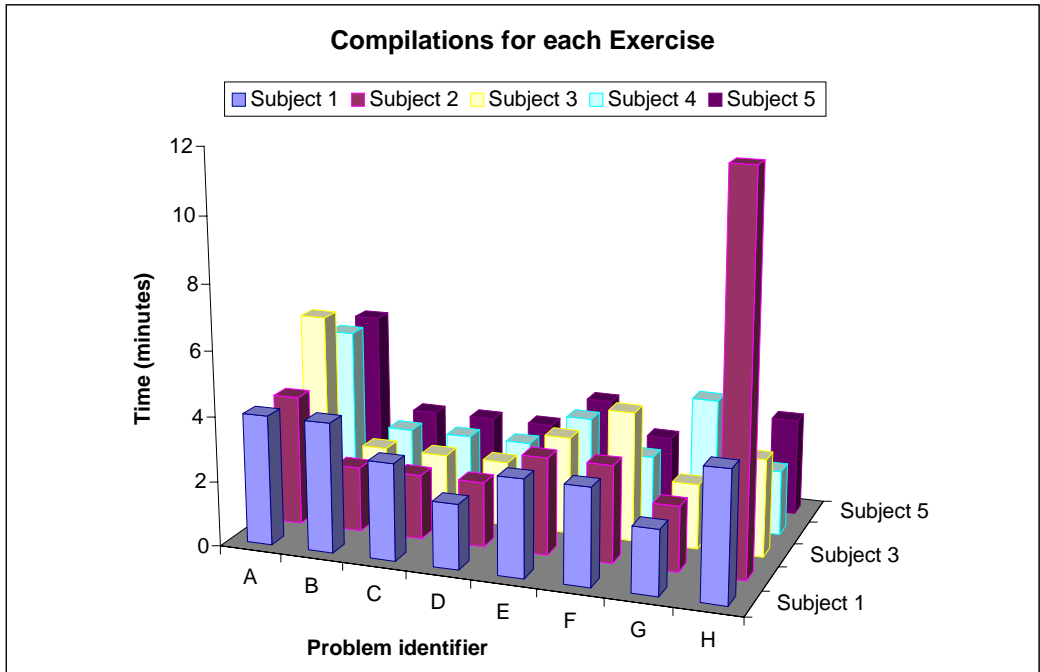
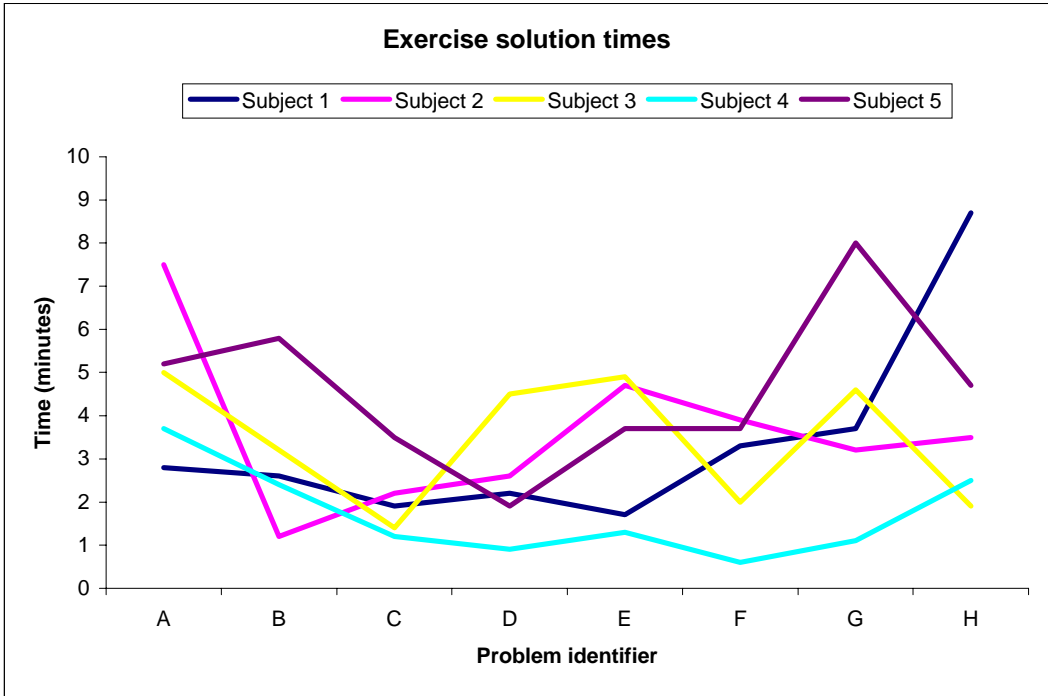
g.cc

Looks at the code then compiles. The declaration looks valid. Searches for the Node struct. Looks at the reference guide. Asks for clarification of the explanation. Goes back to the code, correcting the problem by commenting the declaration of int Node.

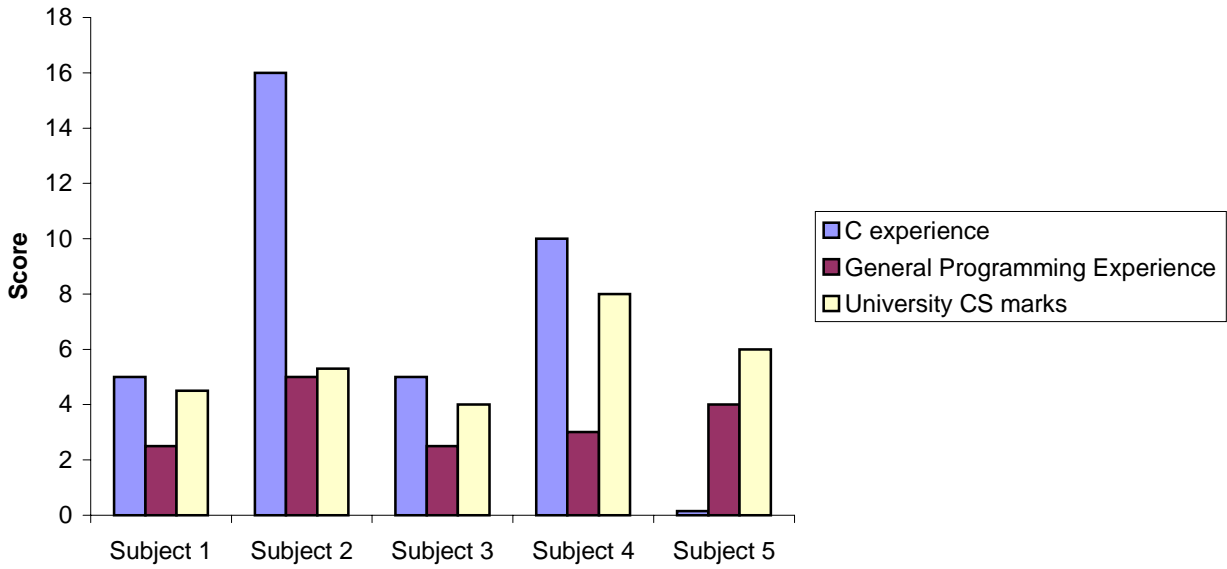
h.cc

Reads the error, goes to the line number, checks for the correct function name according to the error message. Reads the line and fixes the problem by putting parentheses around the addition. Then looks at the reference guide for confirmation. Comments “you are making an assignment, you probably wanted to do a test”. Looks only at the line indicated by the error, deletes stream.

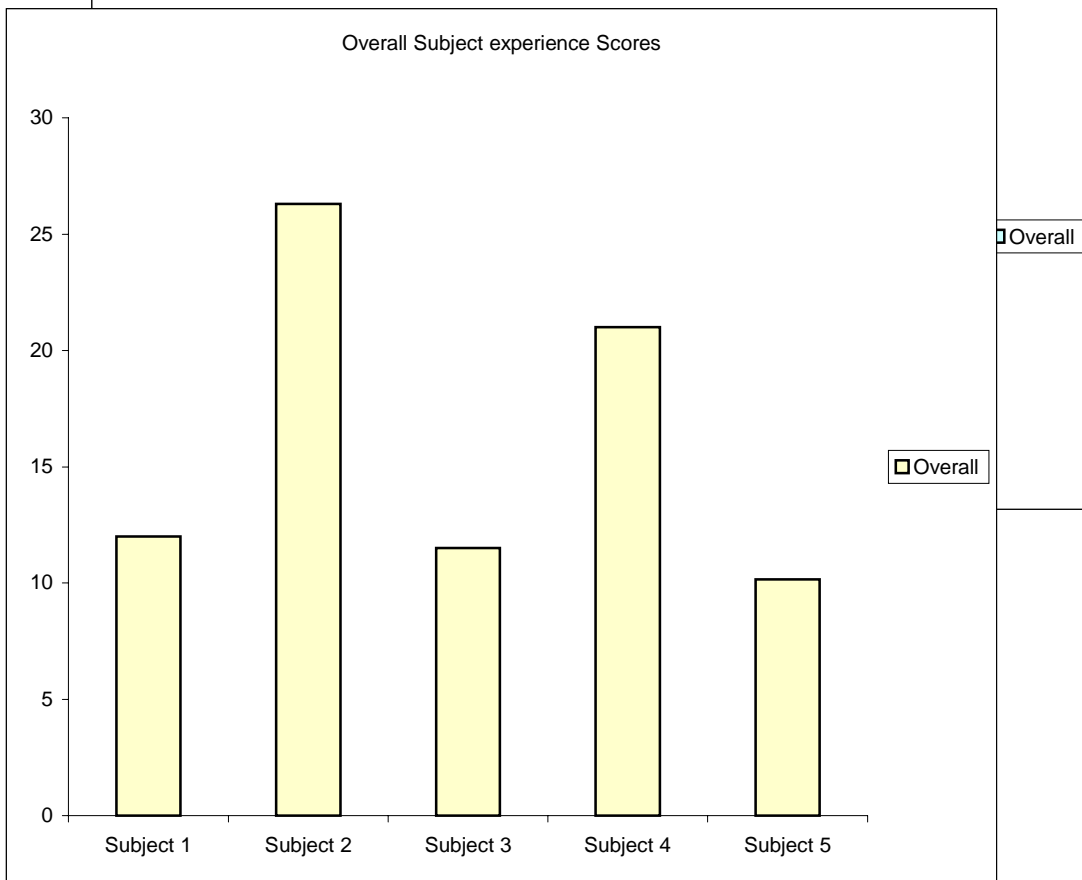




Subject Experience Statistics



Overall Subject experience Scores



Appendix C: Reference Guide for g++ compile errors

parse error before `>`

'*' in wrong location	
<pre>vector < *int > nums;</pre>	<pre>vector < int* > nums;</pre>

extraneous `char` ignored

parse error before `*`

If the semi-colon after a struct or class is missing, strange errors can occur after that point.	
<pre>class thing { int x; } char * function() { // code }</pre>	<pre>class thing { int x; }; char * function() { // code }</pre>

parse error before `word`

Missing open comment, /*	
<pre>person * db[10]; The symbol table data base */</pre>	<pre>person * db[10]; /* The symbol table data base */</pre>

semicolon missing after declaration of `person`

Struct or class missing the ending semicolon.	
<pre>struct person { int x; } char *</pre>	<pre>struct person { int x; }; char *</pre>

```
function(char * input)
{
    return input;
}
```

```
function(char * input)
{
    return input;
}
```

parse error before `{'

A missing brace causes an error. However, the line number indicated is at the beginning of the next function making it difficult to identify.

```
int
main()
{
    int x;
    while(x < 10)
    {
        x++;
        cout << x << endl;
    }

int function(int x)
{
    x = x + 1;
    return x;
}
```

```
int
main()
{
    int x;
    while(x < 10)
    {
        x++;
        cout << x << endl;
    }

int function(int x)
{
    x = x + 1;
    return x;
}
```

'var_name' undeclared (first use in this function)
(Each undeclared identifier is reported only once
for each function it appears in.)

There are multiple possibilities for this error. A variable is not declared, in this case y.

```
void function()
{
    int x;
    x = 0;
    while(x < 10)
        y = y + x;
}
```

```
void function()
{
    int x;
    int y;
    x = 0;
    while(x < 10)
        y = y + x;
}
```

This error can also be caused by a Variable is given the same name as a class or struct

<pre>class person { int height; int age; }; int main() { char * person; person * curr_people[100]; }</pre>	<pre>class person { int height; int age; }; int main() { char * person_string; person * curr_people[100]; }</pre>
---	--

type specifier omitted for parameter

'var_name' undeclared (first use in this function)
(Each undeclared identifier is reported only once
for each function it appears in.)

The second argument is missing its type.	
<pre>int function(int a, b) { int ans = a*a - b*b; ans += 1; return ans; }</pre>	<pre>int function(int a, int b) { int ans = a*a - b*b; ans += 1; return ans; }</pre>

'cout' undeclared (first use this function)
(Each undeclared identifier is reported only once
for each function it appears in.)

The iostream library was not included.	
<pre>int main() { cout << "test printing\n"; }</pre>	<pre>#include <iostream> int main() { cout << "test printing\n"; }</pre>

ANSI C++ forbids declaration 'function' with no type

The function is missing a return type.	
---	--

```
function(int a, int b)
{
    int ans = a*a - b*b;
    ans += 1;
    return ans;
}
```

```
int function(int a, int b)
{
    int ans = a*a - b*b;
    ans += 1;
    return ans;
}
```

conflicting types for `thing`
previous declaration of `thing`

The variable has been redeclared with a different type

```
int main()
{
    int thing;
    thing = 10;
    char thing;
    thing = 's';
}
```

```
int main()
{
    int thing;
    thing = 10;
    char thing2;
    thing2 = 's';
}
```

too few arguments to function `int function(int, int)`
at this point in file

The function prototype and its declaration have different numbers of arguments

```
int function(int, int);

int main()
{
    int current;
    current = function(10);
}

int function(int x)
{
    x = x + 1;
    return x;
}
```

```
int function(int);

int main()
{
    int current;
    current = function(10);
}

int function(int x)
{
    x = x + 1;
    return x;
}
```

warning: suggest parentheses around assignment used as truth value

--	--

<p>used instead of "==" in an if statement. There is no error because, if(x = 10) is valid, it's just always true (since it is an assignment). It is a very common mistake and a good warning to remember.</p>	
<pre>void function(int x) { if(x = 10) cout << "big\n"; }</pre>	<pre>void function(int x) { if(x == 10) cout << "big\n"; }</pre>

warning: useless keyword or type name in empty declaration

<p>This warning can mean that a typedef is missing its 'new' name. It is a common practise to provide typedefs for structs so that you don't have to type 'struct thing' every time you want to refer to the struct. Instead you can just type 'thing'.</p>	
<pre>typedef struct box;</pre>	<pre>typedef struct box box;</pre>

warning: control reaches end of non-void function

<p>Forgetting to return anything from a function that is suppose to return something (i.e. has a return type which is not void) will cause this warning.</p>	
<pre>int function() { x = 10; }</pre>	<pre>void function() { x = 10; }</pre>
	<pre>int function() { x = 10; return x; }</pre>
<p>This can also occur if the return types are nested inside conditional (if/case/loops) statements - meaning that it is possible to reach the end of the function without returning anything.</p>	
<pre>int function(int x)</pre>	<pre>void function(int x)</pre>

<pre>{ if(x == 4) return 1; }</pre>	<pre>{ x = 10; }</pre>
	<pre>int function(int x) { if(x == 4) return 1; return 0; }</pre>

warning: return-type defaults to `type`

<p>If the semi-colon after a struct or class is missing, strange errors can occur after that point. This warning is one of those weird errors.</p>	
---	--

return type for `main` changed to `int`

<p>main() is expected to return an integer.</p>	
<pre>void main() { // code }</pre>	<pre>int main() { // code }</pre>

Appendix D: Programming Experience Questionnaire

Part A: Programming Experience

1. How long have you been programming (approx # of years)?
2. Which programming languages are you familiar with?
3. If you took COMP1001/1901, when was it (which year) and what was your grade (F, CP, P, C, D, HD)?
4. If you took COMP1002/1902, when was it (which year) and what was your grade (F, CP, P, C, D, HD)?
5. If you took DDS, when was it (which year) and what was your grade (F, CP, P, C, D, HD)?
6. If you took PPU, when was it (which year) and what was your grade (F, CP, P, C, D, HD)?
7. If you took OOS, when was it (which year) and what was your grade (F, CP, P, C, D, HD)?
8. If you took Software Engineering, when was it (which year) and what was your grade (F, CP, P, C, D, HD)?
9. Please list any other programming courses you have taken.

Part B: Previous Experience with C/C++

10. How long have you been programming in C/C++?
11. List any major projects you have completed in C/C++ (include assignments).

Part C: The reference that I provided

12. For those of you who were given the chance to use the reference guide, did you find it helpful (btw, it is not going to upset me if you all say it was useless, so please be honest)?
13. Given access to this type of reference (with additions to make it more comprehensive), how likely do you think you would be to use it for future programming projects.

Part D: Any other comments