GIYF: INVESTIGATING AND REMOVING BARRIERS TO STUDENT SELF SUFFICIENCY

BY

SHOAIB MEENAI

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Associate Professor Craig Zilles
Abstract

Student self sufficiency is essential for Computer Science students to be successful in the field. It enables them to work through their problems independently and keep up with rapidly changing development ecosystems, and is a significant contributor to success both as a student and in the workplace.

Student self sufficiency is especially important as the demand for Computer Science keeps rising and rising. Self sufficient students place a significantly smaller load on difficult to scale course resources such as office hours, and encouraging student self sufficiency enables courses to better cope with larger enrollments.

To identify common problems students encounter when they attempt to solve their problems on their own, we created test instruments to mimic common problems students may encounter in introductory courses. We then administered these instruments to eleven students and recorded their thought process for analysis.

We discovered a number of common barriers faced by students in the pursuit of self sufficiency. These barriers indicate several common underlying problems. Future work can focus on addressing these problems in a systematic manner.
To my mother, whose voice can always cheer me up no matter how terribly my day is going.
Acknowledgments

I am deeply grateful to my adviser Craig Zilles for giving me the freedom to explore a research area which was close to my heart, helping and guiding me throughout the process, and constantly inspiring me to become a better educator, mentor, and human being.

I would like to thank Chase Geigle and Cinda Heeren for their help with recruiting students for the study. I would also like to thank Cinda for reviewing the test instruments and offering suggestions.

Thanks to all the students who participated in my research and allowed their responses to be used. I gained a lot of valuable insight from them.

A huge thank you to my friends and family, who supported me throughout and served as guinea pigs for some of the instruments. I am especially thankful to Purajit, who generously let me stay at his apartment while I was working on this.
# Table of Contents

Chapter 1  Introduction .................................................................................. 1  
  1.1  Motivation ......................................................................................... 1  

Chapter 2  Methodology .............................................................................. 3  
  2.1  Test Instruments ............................................................................... 3  
  2.2  Student Interviews .......................................................................... 3  

Chapter 3  Test Instruments ...................................................................... 4  
  3.1  Instrument 1: Missing Include (Error First) ....................................... 4  
  3.2  Instrument 2: Undefined Function (Error First) ............................... 5  
  3.3  Instrument 3: Wrong Access Operator (Error First) ....................... 6  
  3.4  Instrument 4: Variable Length Arrays (Code then Error) ............. 7  
  3.5  Instrument 5: Signed Comparisons (Code then Error) .................... 8  
  3.6  Instrument 6: Iterator Const Correctness (Code then Error) .......... 9  
  3.7  Instrument 7: Map Const Correctness (Code then Error) ............... 10  
  3.8  Instrument 8: Pointer to Member (Code Understanding) .............. 12  
  3.9  Instrument 9: Deduplication using Sets (Code Writing) ............... 13  
  3.10 Instrument 10: Missing Brace (Code then Error) ......................... 14  

Chapter 4  Results ....................................................................................... 17  
  4.1  Instrument 1: Missing Include ....................................................... 18  
  4.2  Instrument 2: Undefined Function ................................................ 18  
  4.3  Instrument 3: Wrong Access Operator .......................................... 19  
  4.4  Instrument 4: Variable Length Arrays .......................................... 19  
  4.5  Instrument 5: Signed Comparisons ................................................ 20  
  4.6  Instrument 6: Iterator Const Correctness ...................................... 20  
  4.7  Instrument 7: Map Const Correctness ........................................... 20  
  4.8  Instrument 8: Pointer to Member ................................................... 21  
  4.9  Instrument 9: Deduplication using Sets ........................................ 22  
  4.10 Instrument 10: Missing Brace ...................................................... 23  

Chapter 5  Conclusions and Future Work .................................................. 24  
  5.1  Student Self Sufficiency ................................................................. 24  
  5.2  Future Work .................................................................................... 24  
  5.3  Conclusion ...................................................................................... 25  

Appendix  IRB consent form .......................................................................... 26  

References .................................................................................................. 27
Chapter 1

Introduction

Computer Science is a field that has been in high demand for the past few years. Demand has far outstripped supply, leading to a steep increase in the number of students applying to study Computer Science at universities across the nation. Consequently, enrollment in Computer Science courses has also increased dramatically, and scaling courses rapidly comes with its own set of challenges. [1,2]

In this thesis, I address the topic of student self sufficiency, i.e., the ability of students to solve their problems on their own, specifically in the context of an introductory Data Structures course. Self sufficiency is vital in Computer Science, both as a student and an industry professional. This thesis takes the first step towards actually developing a formal set of instructional materials addressing self sufficiency, by investigating common problems students encounter when solving problems.

This research was conducted at the University of Illinois at Urbana-Champaign, which has also seen a rapid increase in the number of students accepted to Computer Science over the past few years. Specifically, it targets students in CS 225, which is the introductory Data Structures course and one of the most fundamental courses in the entire curriculum. Both Computer Science and Computer Engineering majors are required to take this course, and many students in other engineering and science disciplines often take it as an elective.

1.1 Motivation

Self sufficiency is an extremely important skill to develop. Courses offer resources such as office hours and Piazza newsgroups to allow students to get their questions answered from course staff. However, it is difficult to scale these resources to increasing course sizes, and office hours in particular are often overwhelmed with the sheer volume of students seeking help [3]. A self sufficient student can work out their problems by themselves, or with a small amount of guidance in the right direction, rather than having to wait multiple hours to receive detailed assistance from course staff.

The increasing interest in Computer Science also results in many students coming into courses lacking prior experience in the field, especially students belonging to traditionally underrepresented groups [4]. Self sufficiency is an effective way for such students to catch up to their more experienced peers. A self sufficient student can gain important skills on their own time, whereas a student heavily reliant on formal instruction risks falling behind.

Many students enter Computer Science and related fields because of a strong interest in programming and software engineering. Software engineering is a relatively young field. New programming languages, frameworks and best practices are constantly being developed [5-7], and it is simply impossible to specifically teach all the possible languages and frameworks a student might encounter. Instead, a self sufficient student, once taught a base set of skills, can successfully navigate the myriad of available options themselves, and quickly learn a new set of tools and skills when needed.

Self sufficiency is especially vital for students who wish to enter the industry. Many tech companies operate in a fast-paced, hands-off environment, and being able to quickly get up to speed with internal tools and development practices is required to be successful. These companies simply may not have enough
resources to provide detailed instruction to newcomers, and so being a self-sufficient and self-directed learner is a huge asset. This is especially true of startups and small companies, but even more established companies such as Facebook have a philosophy of moving fast [8], and moving fast requires learning fast.

Finally, on a more personal level, I have been a Teaching Assistant for an introductory Computer Architecture course for seven semesters, and have taught it one summer. I have encountered numerous instances of students asking questions that could have been answered via a simple Google search, to the point where the phrase Google Is Your Friend (and variations with the same implication) has become a common mantra of mine. I have instructed both high performing and struggling students, and I have become convinced through these experiences that self sufficiency is a large part of what differentiates the two groups.

Despite the importance of student self sufficiency, it is not formally taught in a course setting. Although formal teaching may seem to run counter to the entire point of self sufficiency, the idea is that there are a certain set of techniques that students can use to direct their own learning, and these techniques greatly increase the efficiency and efficacy of students when self learning. Presently, the only way for students to learn these techniques is on an individual basis, through their own experience or their encounters with their peers and course staff. This will obviously not scale well with increasing course enrollments, and so more efficient dissemination techniques are needed.

This thesis represents the first step in developing such techniques. A set of test instruments was developed and administered to students to identify the common causes of students failing to be self sufficient. Chapter 2 discusses the testing methodology, Chapter 3 details the test instruments used, and Chapter 4 analyzes the results. Finally, Chapter 5 discusses how our discoveries can be used to effectively tackle the problem of student self sufficiency.

A literature search yielded surprisingly few instances of similar research. Bloom and Fekete [9] discuss redesigning a Programming Languages course to put self sufficiency and critical thinking at the forefront, but their work is specific to that particular setting. Radenski [10] applies abductive learning principles to an introductory computer science course, but their work is more of a general pedagogical technique, whereas ours chooses to focus specifically on a narrow area (mostly searching to diagnose compiler errors).
Chapter 2

Methodology

2.1 Test Instruments

In order to understand common impediments that students encounter when attempting to solve their problems on their own, we devised a number of test instruments. The idea behind these instruments was to test simplified versions of actual problems that students might encounter in their introductory Data Structures course. To this end, the questions were drawn both from my experiences of the common problems that students ran into in my role as a teaching assistant, and from frequently asked questions on the Piazza newsgroup for the Data Structures course.

The instruments were then reviewed by the veteran instructor for the Data Structures course to ensure that they met the desired objective of testing realistic and frequent problems, and revised accordingly. Chapter 3 contains a detailed discussion of each of the chosen instruments.

2.2 Student Interviews

Once the instruments had been devised, they were administered to students, who were asked to work through the instruments and voice their thought process out loud. Each of the student interviews lasted for one hour, and the students were remunerated $20 for their participation. Detailed notes of student behavior were made for future review and analysis. Since the research involved human subjects, IRB approval was required; the informed consent form provided to each participant is included in the Appendix.

The test instruments were administered to eleven students. Two of the students had recently completed the Data Structures course, while the remaining nine were currently enrolled in that course (and at the time of administration, the course was just past its halfway point). The students were recruited through an email advertisement sent out to the entire class, and then filtered to ensure a good distribution of majors, genders and class standing. Detailed demographics follow.

The results of the interviews are discussed in detail in Chapter 4.

<table>
<thead>
<tr>
<th>Computer Science</th>
<th>Computer Engineering</th>
<th>Other (Engineering)</th>
<th>Other (Science)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.1: Participant major distribution

<table>
<thead>
<tr>
<th>Female</th>
<th>Male</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2.2: Participant gender distribution

<table>
<thead>
<tr>
<th>Sophomore</th>
<th>Junior</th>
<th>Senior</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.3: Participant class standing distribution
Chapter 3

Test Instruments

In order to understand how students deal with problems that might arise in an introductory course setting, a number of test instruments were administered. These are detailed below, with an explanation of what each instrument was designed to test. We used four types of instruments:

1. **Error First**: Students were first shown an error message and then asked to state all the information they could glean from it and what a probable cause of the error could be. After they did so, they were shown the actual code to confirm their guess.

2. **Code then Error**: Students were first asked to read through a code snippet which contained a non-obvious compile error. The code was then compiled and students were shown the error message and asked to determine the cause.

3. **Code Understanding**: Students were asked to read through a code snippet which used a language feature they would be unfamiliar with. They were then asked to determine the code’s behavior.

4. **Code Writing**: Students were shown an incomplete piece of code. It was explained to them what the code was meant to accomplish, and then they were asked to complete the code according to the given specification.

### 3.1 Instrument 1: Missing Include (Error First)

The error message was as follows:

```
q1.cpp:2:5: error: use of undeclared identifier 'cout'
    cout << "Hello world!\n";
   ^
```

Error messages contain a lot of information. There’s the file name and line number (and column number), q1.cpp:2:5 in this case, which is the location of the error. There’s the actual error message, which is fairly simple here, but can often become cryptic and contain a lot of program-specific information which would need to be filtered out when searching. Modern compilers also display the erroneous line of code, with a caret underneath pinpointing the exact location of the error.

The purpose of this instrument was to understand which of these pieces of information students pay attention to and in what order they process the message. The error itself is very straightforward and something most students would have encountered themselves, so it was expected for them to be able to figure out the cause from just the error message alone. The erroneous code was the following:

```cpp
int main() {
    cout << "Hello world!\n";
}
```

**Listing 1: q1.cpp**
The error here is that the iostream header, which defines the cout object, is not included, and therefore cout is unrecognized. Furthermore, cout is part of the std namespace, so it would either have to be qualified or a using directive used.

3.2 Instrument 2: Undefined Function (Error First)

The error message was as follows:

Undefined symbols for architecture x86_64:
"foo::unnecessarily_long_function_name()", referenced from:
    _main in q2-722e57.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)

This instrument tests a linker rather than a compiler error, and consequently the error message is a bit more difficult to parse, and in particular contains no location information. It was expected for students to not be familiar with this particular error, so that they would have to search to figure out a possible cause. The purpose of this instrument was to understand what students searched for and whether they were able to pick out relevant parts of the error message and exclude program-specific ones. Once they had searched, they were shown the actual code:

```cpp
#ifndef Q2_H
#define Q2_H

class foo {
    public:
        void unnecessarily_long_function_name();
};
#endif
```

Listing 2: q2.h

```cpp
#include "q2.h"

int main() {
    foo f;
    f.unnecessarily_long_function_name();
}
```

Listing 3: q2.cpp

The function unnecessarily_long_function_name is declared but not defined, hence the linker error. Students were expected to be able to understand the general cause of the error based on their searches.
3.3 Instrument 3: Wrong Access Operator (Error First)

The error message was as follows:

q3.cpp:7:14: error: member reference type 'string' (aka 'basic_string<char, char_traits<char>, allocator<char> >') is not a pointer; maybe you meant to use '.'?
    cout << a->length() << endl;
   ^

The error message actually contains the exact cause of the error and suggests a fix, but it also has some noise, particularly in the template expansion:

basic_string<char, char_traits<char>, allocator<char> >}

The purpose of this instrument was to see if students could understand the error message and suggested fix without getting confused by the template expansion. The actual file is as follows. As said in the error message, the fix would be to use . instead of ->.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string a = "Hello world";
    cout << a->length() << endl;
}
```

Listing 4: q3.cpp
3.4 Instrument 4: Variable Length Arrays (Code then Error)

The code consists of a function which takes an integer parameter and declares an array of that length:

```cpp
#include <iostream>
using namespace std;

void foo(int n) {
    int arr[n];
    arr[0] = 42;
    cout << "I made an array of size " << n << endl;
}

int main() {
    foo(55);
}
```

Listing 5: q4.cpp

The resulting error message is as follows:

```
q4.cpp:5:12: error: variable length arrays are a C99 feature [-Werror,
-Wvla-extension]
    int arr[n];
    ^
```

The error message points out the source of the error, and it was expected for most students to be able to guess the cause based on that and their prior experience with C++. However, the error message is also somewhat indirect. The purpose of this instrument was to see if students would be able to figure out the cause of the error directly, and if they would be curious about the meaning of the error message (in particular, the “are a C99 feature” portion) and would search to understand that.
3.5 Instrument 5: Signed Comparisons (Code then Error)

The code attempts to sum up the elements of a vector:

```cpp
#include <vector>
using namespace std;

int sum_vector(const vector<int> & v) {
  int sum = 0;
  for (int i = 0; i < v.size(); ++i) {
    sum += v[i];
  }
  return sum;
}
```

Listing 6: q5.cpp

The resulting error message is as follows:

```bash
q5.cpp:6:23: error: comparison of integers of different signs: 'int' and 'size_type' (aka 'unsigned long') [-Werror,-Wsign-compare]
  for (int i = 0; i < v.size(); ++i) {
    ~ ^ ~~~~~~~~
```

As the message says, the error lies in the comparison, and it was expected that students would be able to figure out the cause of the error and how to fix it from the message alone. However, they would then be asked why the error occurs, i.e., why comparing signed and unsigned numbers produces an error, since it’s mathematically possible to compare negative and positive numbers.

The purpose of this instrument was to see what students searched for to attain this higher level of understanding. Students were not necessarily expected to have been exposed to 2’s complement representation, which is the reason for the error – the C++ standard mandates that signed numbers be converted to their unsigned equivalent before comparing to an unsigned number, which means \(-1 < 10u\) would actually be false – so it would test how they process unfamiliar topics and filter out information that’s irrelevant or too specific.
3.6 Instrument 6: Iterator Const Correctness (Code then Error)

The code also attempts to sum up the elements of a vector, but using iterators instead of an index loop:

```cpp
#include <vector>
using namespace std;

class VectorWrapper {
  public:
    VectorWrapper(vector<int> v) : v(v) {
      // empty
    }

    int sum() const {
      int sum = 0;
      for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
        sum += *it;
      }
      return sum;
    }

  private:
    vector<int> v;
};
```

Listing 7: q6.cpp

The resulting error message is as follows:

```
q6.cpp:12:40: error: no viable conversion from '__wrap_iter<const_pointer>'
to '__wrap_iter<pointer>'
  for (vector<int>::iterator it = v.begin(); it != v.end(); it++) {
      ~~~~~~~~~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/iterator:1116:7: note: candidate constructor
the implicit copy constructor) not viable: no known conversion from
'const_iterator' (aka '__wrap_iter<const_pointer>')) to
'const std::__1::__wrap_iter<int *> &' for 1st argument
  class __wrap_iter
^  ~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/iterator:1116:7: note: candidate constructor
the implicit move constructor) not viable: no known conversion from
'const_iterator' (aka '__wrap_iter<const_pointer>')) to
'std::__1::__wrap_iter<int *> &&' for 1st argument
```
class __wrap_iter
   ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain
/usr/bin/../include/c++/v1/iterator:1239:31: note: candidate constructor not
viable: no known conversion from 'const_iterator' (aka '__wrap_iter<
const_pointer>') to 'iterator_type' (aka 'int *') for 1st argument
_LIBCPP_INLINE_VISIBILITY __wrap_iter(iterator_type __x) _NOEXCEPT :
   __i(__x) {} ^
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain
/usr/bin/../include/c++/v1/iterator:1138:28: note: candidate template ignored:
disabled by 'enable_if' [with _Up = const int *]
   typename enable_if<is_convertible<_Up, iterator_type>::value>::type* =
   0) _NOEXCEPT
   ^

This was the first long error message the students were exposed to; it hints at the cause, which is that an
iterator is being used to access a const vector, which instead requires the use of a const_iterator,
but also contains detailed information about why the conversion failed, which can be overwhelming for
novices. The purpose of this instrument was to see how students dealt with the sheer volume of information
in the error message, and if they struggled with separating the general portions of the error from the
program-specific ones.

3.7 Instrument 7: Map Const Correctness (Code then Error)

The code defines a template function for accessing the 0th indexed element of a container:

```cpp
#include <iostream>
#include <map>
using namespace std;

template <class T>
int first_element(const T& container) {
   return container[0];
}

int main() {
   map<int, int> m;
   m[0] = 32;
   cout << first_element(m) << endl;
}
```

Listing 8: q7.cpp
This is the resulting compile error:

```cpp
q7.cpp:7:12: error: no viable overloaded operator[] for type 'const std::map<int, int, std::less<int>, std::allocator<std::pair<const int, int> > >'
    return container[0];
^~~~~~~~~
q7.cpp:13:13: note: in instantiation of function template specialization
  'first_element<std::map<int, int, std::less<int>, std::allocator<std::pair<const int, int> > > >' requested here
    cout << first_element(m) << endl;
^/
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/map:1006:18: note: candidate function not viable: 'this' argument has type 'const std::map<int, int, std::less<int>, std::allocator<std::pair<const int, int> > >', but method is not marked const
    mapped_type& operator[](const key_type& __k);
^/
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/map:1008:18: note: candidate function not viable: 'this' argument has type 'const std::map<int, int, std::less<int>, std::allocator<std::pair<const int, int> > >', but method is not marked const
    mapped_type& operator[](key_type&& __k);
    ^
```

This is another long error message, and it also contains a full template expansion of a standard type (std::map<int, int> in this case). The message does contain the cause, which is that the [] operator on maps is not const (which necessitates the use of the at function instead), and it was expected for at least some students to be able to guess the error based on a careful reading of the message alone. However, they would then be asked why the [] operator was not marked const, i.e., why it couldn’t be used for just reading an element from a const map. The purpose of this instrument was to understand what students searched for to determine the answer, how direct they were in their queries, and how successfully they could read a large documentation page and pick out the relevant parts.
3.8 Instrument 8: Pointer to Member (Code Understanding)

The code is as follows:

```cpp
#include <iostream>
using namespace std;

class Hmm {
    public:
        int x;
};

int main() {
    int Hmm::*px = &Hmm::x;

    Hmm h;
    h.*px = 42;
    cout << h.x << endl;
}
```

Unlike the prior instruments, this code actually compiles and runs perfectly. It uses an arcane feature of C++ known as pointer-to-member variables, which students would not be expected to know. It was expected for some students to be able to guess the program’s behavior just from the code, but then they would be asked to determine what exactly the code was using and how it worked, which would necessitate a search.

The purpose of this instrument was to see how students attempted to search for syntax, particularly since search engines usually don’t handle symbols very well. There are many commonly used operators and language features which are difficult to search for by symbols alone if one is unfamiliar with their names, e.g., the ternary operator, the null-conditional operator in C#, and list comprehensions in Python, so although this particular instance is somewhat contrived, the general problem is one that novice programmers would encounter fairly often.
3.9 Instrument 9: Deduplication using Sets (Code Writing)

Students were presented with the following code shell:

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<int> deduplicate(const vector<int> & nums) {
}

int main() {
    vector<int> nums = { 10, 20, 30, 10, 41, 98, 7, 5, 7, 10, 23, 85 };
    vector<int> deduplicated = deduplicate(nums);
    for (size_t i = 0; i < deduplicated.size(); ++i) {
        cout << deduplicated[i] << " " << endl;
    }
}
```

Listing 10: q9.cpp

They were then asked to write the `deduplicate` function, which should take a vector of integers and return a new vector, which contains one copy of each distinct element from the original vector, with the order of the new vector being insignificant. They were first asked to describe an algorithm; the desired algorithm was to use a `set` to filter out duplicates, and students were guided to that algorithm. Once they reached it, they were asked to actually write the code and then compile and run it to verify its correctness.

The purpose of this instrument was twofold. Firstly, it was to understand a student’s thought process when coming up with an algorithm to tackle a given task, which of course is a vital part of programming. Secondly, students were not expected to have used the `set` class before, so the instrument would test how students went about determining that a `set` was the optimal data structure to use, and how successfully they could use class documentation to figure out how to use a new class.
3.10 Instrument 10: Missing Brace (Code then Error)

Students were shown the following C++ file:

```cpp
#include "q10.h"
#include <iostream>
using namespace std;

int main() {
    cout << "Hello world!" << endl;
}
```

Listing 11: q10.cpp

The resulting error messages are:

In file included from q10.cpp:2:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/iostream:38:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:216:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/__locale:15:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:436:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain
/usr/bin/../include/c++/v1/cstring:69:7: error: no member named 'size_t' in
the global namespace; did you mean '::__clang__internal::size_t'?
using ::size_t;
  ^~
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain
/usr/include/sys/_types/_size_t.h:30:32: note: '::__clang__internal::size_t' declared here
typedef __darwin_size_t size_t;
^}
In file included from q10.cpp:2:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/iostream:38:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:216:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/__locale:15:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains
/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/string:436:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain
/usr/bin/../include/c++/v1/cstring:70:7: error: no member named 'memcpy' in
14
the global namespace; did you mean '::__clang__internal::memcpy'?
using ::memcpy;
^~
/usr/include/string.h:72:7: note: '::__clang__internal::memcpy' declared here
void *memcpy(void *, const void *, size_t);
^

<many similar errors omitted for brevity>

In file included from q10.cpp:2:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:38:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:216:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:216:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:216:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:216:
In file included from /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/ios:216:
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/../include/c++/v1/cstring:94:7: error: no member named 'strstr' in the global namespace; did you mean '::__clang__internal::strstr'?
using ::strstr;
^~
/usr/include/string.h:89:7: note: '::__clang__internal::strstr' declared here
char *strstr(const char *, const char *);
^
fatal error: too many errors emitted, stopping now [-ferror-limit=]
Students were expected to be able to observe that there are no errors in the C++ file and hence ask to see the header file:

```cpp
#ifndef Q10_H
#define Q10_H

namespace __clang__internal {

    class foo {
        public:
            void lots();
            void and_lots();
            void and_some_more();
            void functions();
            void nothing();
            void more();
            void to();
            void see();
            void here();
            void move();
            void along();
    }

#endif
```

Listing 12: q10.h

The error here is that the namespace declaration is missing its closing brace, but the number of member functions and the odd-looking name `__clang__internal` were both deliberately chosen to obscure the actual error. The purpose of this instrument was twofold: firstly, to see how students would deal with an overwhelming amount of error messages, and secondly, to see if they would notice that the errors are occurring in system files, which are extremely unlikely to be the actual source of the error, and hence instead peruse the given code carefully and ask to look at the header file.
Chapter 4

Results

We made a number of important general observations from the results of our instruments:

- Students performed better than expected at filtering out program-specific information from error messages before searching for them; in particular, no student included file or function names in their searches. However, on more complicated error messages, students often included specific types in their initial searches, and only thought to filter them out when the initial search yielded unhelpful results.

- Students paid little attention to the file names and line numbers contained in error messages; they often started reading the error message at the description, skipping over the location entirely. They did pay more attention to the actual erroneous lines in error messages which contained them, and in particular to the caret pinpointing the exact character position of the error.

- Students had difficulty dealing with long and unfamiliar messages. They focused on the beginning of the error message, which describes the actual error, but neglected the remainder, which goes into more detail. In some cases, the description is enough to diagnose the problem and the additional details are unnecessary, but in others, the additional details contain important clues about the cause of the error.

- When reading an error message, students fixated on specific terms that they understood and ignored the parts of the error messages that they did not understand. This skipping can hinder the search process.

- When faced with a full page of search results, students had some difficulty picking out the most relevant ones. They displayed a strong preference for StackOverflow links.

- When reading a StackOverflow link, students skimmed the question and paid more attention to the top answer. This resulted in them not realizing that parts of the question were irrelevant to their query, and they were often unable to generalize the answer to their situation. On the other end, when students realized that the question was specific to certain libraries or IDEs, either from the title or from the question description, they ignored the question entirely, even though the answer often still contained generally applicable advice.

- For StackOverflow links, if the top answer did not answer their question, students would open a different search result instead of examining other answers and comments on the same question.

- Students attempted to take the path of least resistance. If an answer was very long or contained code examples, they would either skim it or skip it entirely and try a different search result. This skimming resulted in them missing out on relevant details.

- In general, students were more concerned with fixing bugs than understanding their causes. Once students had solved the error, they were content to move on, and had to be specifically prompted to dig deeper and actually understand the general cause.
• When writing code using a new data structure, students were faster and less error-prone if they focused on usage examples instead of function signatures and descriptions.

• Students were unfamiliar with common Google tricks, such as replacing symbols with their names (since Google filters out symbols), and either using quotes or stripping the leading – when searching for compiler flags such as `-Wsign-compare` (since the leading – is treated as the negation operator otherwise).

A more in-depth discussion of the observations from each instrument follows.

4.1 Instrument 1: Missing Include

As expected, all the students were able to figure out the error from the error message alone, before looking at the actual file. All the students first looked at the error message and then at the erroneous line, and the combination of the two was enough for them to figure out the cause, particularly since many of them had encountered the error before.

What was surprising, however, was how little attention students paid to the location information in the error message (`q1.cpp:2:5`). Only one student specifically read out the location information in their original reading of the error message. The rest were asked if there was any more information in the error message, and only one student responded with the location information. The rest of the students only volunteered the location information after specifically being asked if the error message told them anything about where the error was occurring, with one student going as far as to state that they always ignore the location information.

It is possible that the straightforwardness of the error and the presence of the erroneous line in the error message itself, along with a caret pinpointing the exact location of the error, is the reason for students not noticing the explicit location information. However, the location is important in general, since the lines surrounding the erroneous line often contain important context, so it is worrying to see students ignore that part of the error message so frequently.

4.2 Instrument 2: Undefined Function

Students struggled with working out the general cause of this error message. One anticipated issue was for students to include the function name in their search, leading to unhelpful results. No student actually made this mistake, however; one was about to, but caught themselves before the actual search. All the students except one either searched for “Undefined symbols for architecture x86_64” or “symbol(s) not found for architecture x86_64”, i.e., either the first line of the error or the second-to-last, suggesting they focus on the beginnings and ends of error messages. The one exception was a search for “linker command failed with exit code 1”, which is the very end of the error message, but also very generic.

Students who made a guess before searching showed a tendency to fixate on the parts of the error message they did understand. In particular, many students noticed the word “architecture” in the error message and thought it related to computer architecture and that the code was being compiled for an incorrect platform or operating system. Some students had seen the error message before, where it had been caused by missing libraries in the linker command, and so they automatically assumed it to be the problem here and started
focusing on that. These findings demonstrate the adage that “a little knowledge is a dangerous thing”, where a small amount of prior knowledge gets students started in the wrong direction.

After constructing good search terms (after some light prompting in a few cases), however, students had a difficult time actually interpreting the results. The search results contain many questions asked in the context of certain libraries or IDEs, and consequently the answers contain both general and question-specific advice. Students struggled with differentiating between the two. Most students found a suggested solution in the answer (for example, a missing library, or trying to compile C++ code with a C compiler) and thought it applied to the instrument right away, without pausing to think about whether the solution actually made sense in the context of the instrument. The root cause was that they were paying more attention to the answer than the question and hence unable to filter out the question-specific parts of the answer. On the flip side, a few students read through the question, decided it was too specific, and skipped to the next search result, whereas some of the answers actually contained applicable general advice.

Overall, only two students were able to work out the general cause of the error by themselves; the rest had to be heavily guided to it. The results suggest that students struggle more with actually interpreting search results than forming queries. Many questions and answers have a low signal to noise ratio when the goal is to find a general solution, and students are unable to sift through extraneous details effectively.

4.3 Instrument 3: Wrong Access Operator

Every student was able to get the error just from the message. About half the students read the entirety of the error message, including the suggested fix at the end, while the other half read the first part of the error message (up to “is not a pointer”), then read the erroneous line, and got the error from there. Both approaches work well for this question, although reading the entire message would be more fruitful in the general case.

A few students commented on the template expansion in the error message, but none of them were distracted by it. A general strategy was for students to skip over the parts of the error message they did not understand and focus on the parts they did, and while that strategy works well for this instrument, it may not in the general case, as discussed for the previous instrument.

4.4 Instrument 4: Variable Length Arrays

All the students focused on the erroneous line in the error message. Most of them were aware that C++ did not allow dynamically-sized stack allocated arrays, and they used the location of the caret in the error message as support for their guess. The two students who were unaware of this language fact Googled the error message to determine the cause.

What was more interesting was that, while the error message “variable length arrays are a C99 feature” is somewhat indirect, only one student expressed curiosity about the “C99 feature” portion and looked it up out of their own volition; the other students were content with just fixing the bug. In the interests of time, the other students were not explicitly quizzed about that portion of the error message. It appears to be another case of students ignoring parts of error messages they don’t understand, which is a pragmatic approach but potentially not conducive to long term conceptual understanding.
4.5 Instrument 5: Signed Comparisons

All the students were able to guess that the comparison was to blame, just from the error message alone. However, some students initially guessed that the different sizes of int and size_t were the cause, despite the error message clearly suggesting it was a problem of different signs, not sizes. Changing the int to a long and recompiling yielded the same error, at which point the students realized it was an issue with signs.

Figuring out why the error was occurring turned out to be more difficult. Similar to the previous instrument, all the students were content with merely fixing the error. However, for this instrument, they were explicitly prompted to understand the root cause of the error, because mathematically, comparing signed and unsigned numbers should not cause any problems.

The first common mistake was that all the students barring one included the specific types when formulating their search query (int and size_t in this case), instead of the more general query of signed and unsigned types. Even the specific queries actually yield useful results in this case, but the general query has the answer in the first result, whereas the specific queries require some more digging.

The second common mistake was to focus on irrelevant parts of a search result and miss the relevant ones. On one of the common StackOverflow questions for this query, the top answer hinted at the behavior of signed and unsigned comparisons being surprising, without explicitly saying how it was surprising. Both the top comment to the top answer and the second answer were explicit, but students just focused on the top answer and had to be guided down. On another common StackOverflow question, the answer contained an example showing the unexpected behavior, but it was also rather wordy. Students skipped over the example, and had to be explicitly told to read and understand it before realizing it was the answer to their query.

These results are similar to Instrument 2: although the queries here were not as general as they should have been, they still yielded useful results, but then students struggled to actually interpret those results. Focusing too heavily on top answers and skipping over code examples are the specific common problematic behaviors exposed by this instrument.

4.6 Instrument 6: Iterator Const Correctness

As expected, most of the students were confused by the length of the error message, with several specifically remaking on it. However, they were eventually able to focus on the top part of the error message, which hints at the cause, though some needed to be prompted to read the error message more carefully before they noticed it. The students who focused on the top part without prompting ignored all the details below it, however, with one even calling them “useless”; while they are not too helpful in this specific case, they can be in general.

Once students found the relevant part of the error message, they were able to diagnose and correct the problem without any further problems. Some of them were unfamiliar with the concept of const iterators and needed to look them up, but they were able to do so by themselves without needing assistance.

4.7 Instrument 7: Map Const Correctness

Students had a much more difficult time with this instrument than anticipated. As with the previous instrument, students focused on the top part of the error message, which states that there’s no viable overloaded operator[], and not on the rest, which explains why none of the overloads are viable. Consequently,
many students thought that a map simply didn’t have an `operator[]` function, and had to look up C++ documentation to convince themselves otherwise after prompting.

Once it was suggested to students that they look over the entire error message carefully, they tended to fixate on the `const` part of the error message. This was indeed the root cause of the problem, but their suggested solution was to either remove the `const` from the parameter of the function or add it to the return type. The former would work but was not in the spirit of the instrument, while the latter would have no effect, which they realized when they actually tried adding it and got a new compile error.

At this point, some of the students realized that `operator[]` would not work on a `const` map, whereas others had to search to discover that. A very common mistake was to search for the entire error message, however, including the template expansion, which was of course not helpful. This was somewhat surprising, since the same students had successfully been able to exclude program specific information for Instrument 2. Another common problem was for students to include the `[]` symbols in their search, whereas Google cannot search for symbols; searching for “square brackets” instead was more fruitful, after it was suggested to them.

Finally, once students had determined the root cause of the error, they were asked why `operator[]` would not work on a `const` map. Most students approached this by looking up the documentation of `operator[]`, which actually explains that the operator creates a key if it does not exist in the function description. However, only one student actually spotted that on their first read; the rest skimmed over the description, skimmed over the rest of the documentation, and were still at a loss. They were then prompted to reread the description, and some of them noticed the answer the second time around, but several still failed to spot it. These students either had to be pointed to the specific paragraph or explicitly asked what the operator did in case the key did not exist to realize what the problem was.

The students’ struggles with this instrument are perhaps a result of their inexperience with searching for the root causes of errors, as also evidenced by the previous instruments. Additionally, these results also demonstrate that students often skip over key information when reading documentation, unless their attention is specifically drawn to it.

4.8 Instrument 8: Pointer to Member

None of the students were familiar with pointer-to-member variables, which was expected. A few were skeptical about the code’s syntactic validity, and were surprised when it actually compiled and ran without errors. Funnily enough, when students were asked what they would do to understand the code, some of them said their first recourse would be to ask a TA, which is the antithesis of the self sufficiency goals of this research.

Searching to discover the name and behavior of this feature also proved to be difficult for students, as expected. Many students were initially completely stumped about approaching the search. Some tried searching for the entire line of code at first, which was of course not helpful, since many of the names were program specific.

Next, students decided to focus on the actual behavior of the program. Some were able to guess the entire output, while others at least noticed that the code was seemingly taking the address of a class member, even though classes do not have addresses. Searching for this strange behavior was not fruitful, however, since the search results were entirely too general and not at all related to this specific feature. Only one student was successful at this stage, because they included the phrase “pointer to class member variable” in their
search, while the rest focused on the aberrant behavior instead.

At this point, a few of the students decided that, since they had not encountered this feature before, it was likely to have been introduced in the new C++ standards, and so they looked up all the new features in C++11 and started trawling through them before being told that it was not a new feature. This once again demonstrated the risk of a small amount of knowledge sending students down a completely incorrect path.

Finally, students were explicitly told to focus on the syntactic oddities in the code and distill those down to their most general form. Eventually, they arrived at either `::*` or `.*`, after some coaxing. Searching for those directly was not helpful, however, since Google strips out most symbols from searches; instead, students had to be told to use the names for the symbols (e.g., “double colon asterisk”), at which point they finally attained relevant search results. Even the symbol to word substitution process is somewhat finicky, however; the `::` operator is known as the scope resolution operator, but searching for “scope resolution star” is decidedly more helpful than searching for “scope resolution asterisk”.

These results suggest that, when encountering a new language feature, students struggle to separate the essence of the feature from the program specific parts. Additionally, in this particular case, a syntax query is actually far more effective than a query based on the observed behavior; even though the latter is based on an actual understanding or attempted understanding of the feature, the resulting query is too general to pinpoint the exact feature being used.

### 4.9 Instrument 9: Deduplication using Sets

Since the students were asked to describe their algorithm before writing any code, they all eventually arrived at the $O(n)$ solution which uses a map/hash table to look up and eliminate duplicate elements. They were then told that there was a standard library data structure that was better suited to this particular problem and that they would likely have not used before. About half the students opted to search for some variant of “C++ data structure to remove duplicates”, while the others looked up a list of all C++ data structures and went through them to discover the appropriate data structure. Both sets of students were able to determine that a `set` was the best data structure for this instrument.

The actual code writing portion yielded some interesting observations. Since none of the students had indeed used a `set` before, they had to use the API reference to work out how to accomplish the given task, and there were two of common approaches. The first was a strictly top down approach, where students first looked up the documentation for the constructor, and then looked through the function list to find other useful functions and their documentation, which were usually `insert`, `find`, and the iterator functions. Students following this approach tended to focus on signatures and descriptions more than examples, which resulted in some difficulty in determining appropriate template and function parameters, but also has an important advantage, which is discussed later.

The other common approach was to look through the list of functions first and then skip straight to the documentation for a useful-sounding function, which was usually either `insert` or `find`. Students following this approach tended to skip over the function signature and description and go straight to the usage examples; they based their own code on the examples, which allowed them to code up a solution much faster and with fewer errors.

Both groups usually arrived at the same solution, where they would use one loop to populate a set with all the elements of the source vector, and then another loop to populate the destination vector with all the elements of the set; the set would handle the deduplication, as intended. However, C++ standard library
containers are actually designed to interoperate, and all containers have a constructor variant which takes start and end iterators and constructs the container from them, which results in a succinct three line solution (with one of those lines being the return statement)!

Only one student was able to arrive at the iterator constructor solution on their own; the rest were shown it after they had written up their own solution, and they were all amazed by its brevity. The only way for a student to discover this solution on their own would be to read the constructor documentation and pay attention to the different overloads, instead of just using the first thing which worked. Students who jumped straight to a particular function’s documentation would have no way of knowing about the iterator constructors, so this is a definite advantage for the top down approach.

The top down approach also has its disadvantages, however. Students using it took longer to formulate their answer and struggled with details like template parameters, though that is also related to their focus on the signature and description instead of the examples. It is also true that, despite around half the students opting for the top down approach, only one of them actually spotted the iterator constructor and took advantage of it. In general, the observations emphasize the importance of examples for student learning; it is much easier for students to base their own code off concrete examples rather than abstract descriptions, but it also limits them to the techniques actually covered in the examples.

4.10 Instrument 10: Missing Brace

Because of time constraints, this instrument could only be administered to five out of the eleven students. The reduced sample size was still enough to make some notable observations, however.

All the students expressed surprise at the amount of errors. Four of the five students focused on the content of the error messages rather than their location, and actually attempted a few unfruitful searches, before their attention was drawn to the location of the messages. Once they realized that the errors were originating in system files, however, they all realized that the errors were most likely in our own files rather than the system files. Since the C++ file was seemingly error free, they requested to see the header.

As expected, some of the students were confused by the namespace name __clang__internal and thought it was the cause of the error; this belief was dispelled by changing the namespace name and demonstrating that the same errors still occurred. Once they had gotten past this hurdle, however, they were all able to note the absence of the closing brace and fix the problem.

The results from this instrument reinforced the idea that students pay much more attention to the contents of the error messages than their location. The location turns out to be especially vital for this instrument in particular, but even in general, it is a very useful piece of information, and students not noticing it is problematic.
Chapter 5

Conclusions and Future Work

5.1 Student Self Sufficiency

Student self sufficiency is vital in both academia and industry, as discussed. It allows courses to scale to an increasing number of students without having to proportionally increase staff resources, which can be infeasible. It enables students to learn new programming languages and frameworks on their own, which is vital for such a fast moving industry. Independent, self sufficient students can be more successful with less guidance, which makes them especially valuable resources for institutions which cannot dedicate too many resources to mentoring, such as startups and small companies.

Our work revealed a number of common barriers to student self sufficiency. Since there is no course or any other sort of formal training specifically targeted at making students more self sufficient, they must acquire such techniques on their own or through ad hoc interactions with their peers and instructors. Consequently, we anticipated a wide range of performance on the interviews. However, while some students definitely worked through the instruments faster than others, there were a number of universal or near-universal problems, which suggest specific weaknesses common to novice Computer Science students.

The most general theme underlying each common problem was an ability to quickly comprehend a vast amount of information, such as a page of search results or an in-depth answer, and pick out relevant portions and discard irrelevant ones. To some extent, this just boils down to experience, since a lot of the information that students were sifting through during the instruments was new to them. Ideally, however, they should still have been capable of reading through an answer, making an informed guess about any unknown portions of the answer, and then generalizing or at least identifying any useful information contained therein, but many of them simply did not read carefully enough.

5.2 Future Work

We have identified a number of common impediments that students face when self learning. Future work can focus on confirming and addressing these impediments.

Since we only had a few existing hypotheses when starting the research, many of our instruments tested a combination of abilities. The discoveries from these instruments provide a starting point for future work in this area. New instruments can be devised to specifically target individual problems, in order to understand how it affects students in isolation.

These instruments, and the new instruments derived from them, can also be administered to experts, e.g., exceptionally performing students and industry professionals. Comparing the approach of novices and experts when tackling the same problems can provide valuable insight into what differentiates the two and how we can guide novices into becoming experts.

Finally, we can develop exercises to actually address the problems discovered, and incorporate them into introductory Computer Science courses. These exercises would focus on teaching students how to teach themselves and disseminate useful techniques to the entire class, which is far more efficient than doing so on
an individual basis, which is the current state of affairs. The instruments would then allow us to measure the impact of including such exercises in the required curriculum.

5.3 Conclusion

The work described in this thesis is an important first step in addressing student self sufficiency. We devised test instruments from commonly encountered student problems, and administered these instruments to students to identify common problems. These common problems provide a basis for future work in the field, including instructional exercises specifically aimed to address these problems and increase student self sufficiency.
Appendix

IRB consent form

Self-Sufficiency Investigation Consent Form

Purpose and Procedures:
The purpose of this research is to identify common problems students face when attempting to search the web for programming information. This research is being undertaken by Shoaib Meenai and Professor Craig Zilles of the University of Illinois at Urbana-Champaign.

Students will participate in interviews with the researchers. During these interviews, students will be given some exercises to complete, which require searching the web for information or error fixes. Students will be asked to voice their thoughts while completing the exercise, and notes will be taken of this thought process to identify common impediments. No personal information will be recorded.

Requirements:
All participants must be at least 18 years old.

Participation is Voluntary:
Participation in this research is voluntary. Students may refuse to participate or may discontinue participation at any time. The decision to participate, decline, or withdraw from participation will have no effect on the subject’s grades at, status at, or future relations with the University of Illinois. Compensation will be prorated for the period of participation.

Benefits and Risks:
As a result of participating in these interviews, students may have a better idea about how to go about searching for the solutions to their programming problems. This is a vital skill to acquire, both for future courses and for industry. Students may experience limited, temporary discomfort, such as stress, related to solving problems out loud, but this should not be greater than experienced in typical student life. Since these interviews do not have any effect on the student’s grade for any course, this discomfort should be very limited.

Compensation:
By participating in an interview of approximately one hour, students will be given $20. If the participant elects to end the interview early, compensation will be prorated at $20/hour.

Confidentiality:
Written notes resulting from this interview will be analyzed by only Shoaib Meenai and Professor Craig Zilles. In the event of publication of this research, no personally identifying information will be disclosed. Under no circumstances will the notes be made public.

Whom to Contact with Questions:
Questions about this research should be directed to Craig Zilles (zilles@illinois.edu). Questions about your rights as a research participant should be directed to the UIUC Institutional Review Board at (217)-333-2670 or irb@illinois.edu.

I certify that I have read this form and volunteer to participate in this research study.

Printed name: ____________________________________________
I give my permission to have notes taken for my research session ______ (check here)

Signature: ________________________________________    Date: _________________
References


