Fundamental Concepts

Chapter 2

Slides by Dr. Ralph Tindell

How to Study

- In each chapter, there are exercises and programming projects marked with the symbol Ⓦ.
- Solutions for these exercise/projects are posted on the text author’s website
- You should try as many of the marked exercises as possible and at least one marked programs for each chapter
- **When you are doing this, DO NOT USE A COMPILER – you will not have access to a compiler during an exam**
- After you have done as much as you can on a problem, go to the website and compare your work with the answer.
- If they are different, try to understand why your solution is different and possibly defective.

Expected Background

- You should have taken a previous programming course in which you learned about:
  - Variables and their types
  - Declaration and initialization of variables
  - Arithmetical and logical expressions
  - The assignment statement
  - if-statements, both with and without an else clause
  - Looping constructs: while, do, and for
  - Declaration and use of arrays
  - Declaration, definition and use of functions.

Expected Background

If you do not have a good grasp of the topics described on the previous slide:

- **You should drop this course and either**
  - **take a lower-level course or**
  - **use this semester to review and practice the material from your previous programming course**
- **Why? Students who have taken this course without an acceptable grade will likely not be able to take it again anytime soon.**
- **Reason: limited capacity and the order in which students are considered for registering in the course:**
  - **Students with a sufficient GPA who have not previously taken this course;** *followed by*
  - **transfer students with a sufficient GPA;** *followed by*
  - **repeaters, in order of their grade in the previous Program Design course.**

The C Language

- Some languages are forgiving.

  - The programmer needs only a basic sense of how things work.

  - Errors in the code are flagged by the compile-time or run-time system

  - The programmer can muddle through and eventually fix things up to work correctly.

- The C language is not like that.

  - The C programming model is that programmers know
    - exactly what they want to do; and
    - how to use the language constructs to achieve that goal.

The C Language

- C is "simple" in that the number of components in the language is small
- If two language features accomplish more-or-less the same thing, C will include only one.
- Programmers can pretty much do whatever they want.
- C's type system and error checks exist only at compile-time.
- There is no garbage collector to manage memory.
- Instead the programmer manages "heap" memory manually.
- All this makes C fast but fragile.

Analysis -- Where C Fits

- Because of the above features, C can be hard for beginners.
- A feature can work fine in one context, but crash in another.
- The programmer needs to understand how the features work and use them correctly. .
- On the other hand, the number of features is pretty small. .
- Perhaps the best advice is
  - Be very careful
  - Don't type things in your program you don't understand. .
  - Debugging takes too much time. .
  - Have a mental picture (or a real drawing) of how your C code is using memory. .
- That's good advice in any language, but in C it's critical.

Parts of a Program

```
/*  This is the traditional first program*/
```
Comment

```
#include <stdio.h>
```
A preprocessor directive

```
int main()
{
    printf( "Hello, World! \n" );
    return 0;
}
```
A function call

A function definition

Every C program has a function named “main”. This is where program execution begins.

The Stages of Compilation

Preprocessing

- Lines beginning with #
  - Textual substitution
  - No knowledge of the C language

Lexical Analysis

- Divide text of source file into words and symbols

The Stages of Compilation

Parsing

Acts on the result of preprocessing

Uses the grammar rules of the C language

Identifies words as variables, function calls, operators, etc

Builds a “parse tree” representing the program

Code Generation

Translate parsed ***source code*** into instructions that are executable by the hardware – ***(binary) object code***

Linking

Source code usually includes references to external functions

Example: **`printf`**

To use an external function you must include the header file of the module that contains that function

For **`printf`**, that would **`be stdio.h`**

Thus one of the first lines in your file would be

```
#include <stdio.h>
```

The angle brackets (<…>) indicate that the header file and associated code resides in a built-in system library

You may include header files you created by replacing the angled brackets with double quotes

```
#include "myfuncs.h"
```

Linking

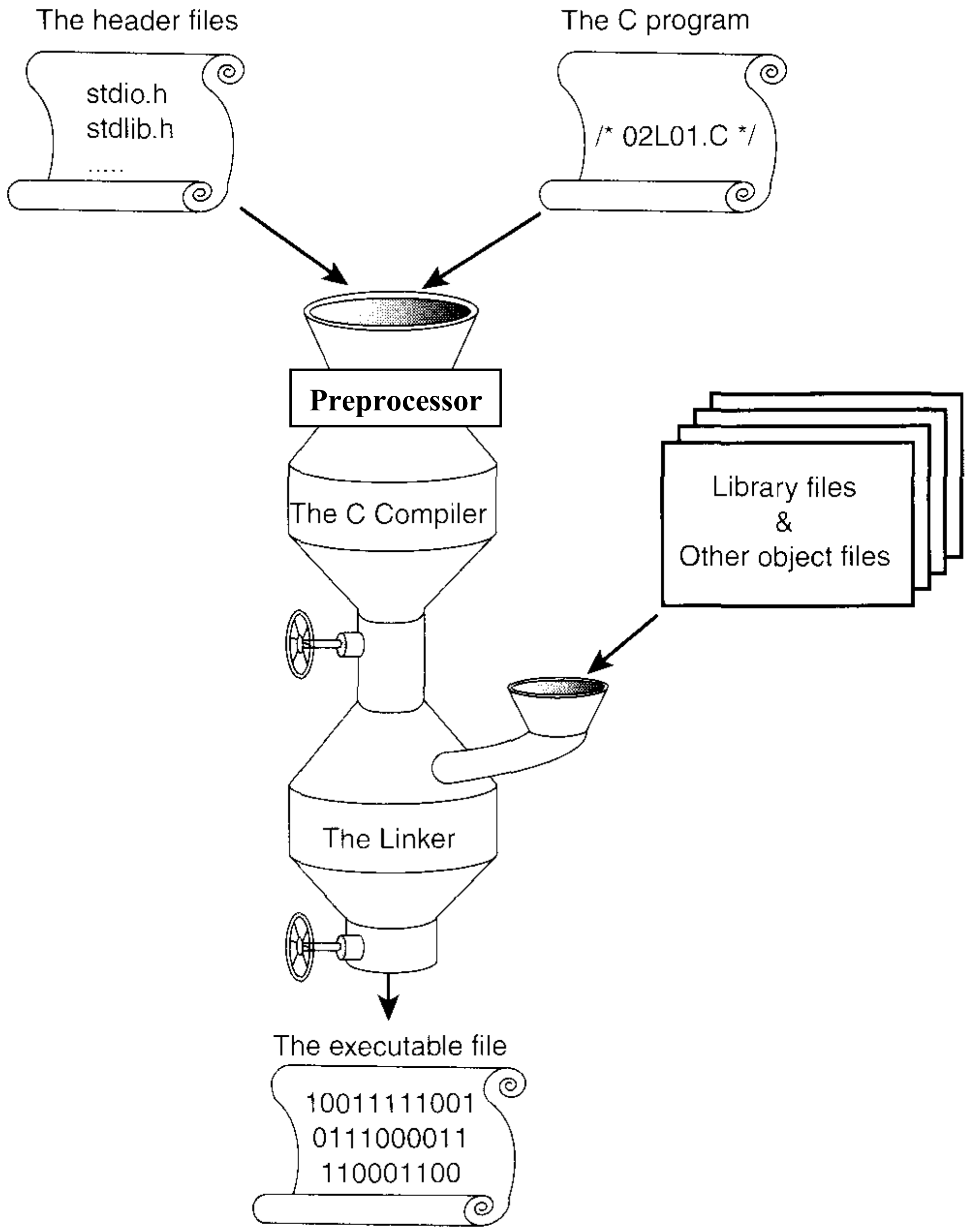
The **linker** produces the executable file by combining

object code produced by the compiler

object code for external functions contained in other modules

The **gcc** command runs a preprocessor, a compiler and a linker (in that order!)

The Compilation Process



Zhang, *Teach Yourself C in 24 Hours*

Preprocessor – First Look

- The preprocessing step happens to the C source before it is fed to the compiler.
- The two most common preprocessor directives are **`#define`** and **`#include`**…

#include

- The "**#include**" directive brings in text from different files during compilation.
- **#include** is very unintelligent and unstructured
- It just pastes in the text from the given file and continues compiling.
- The **#include** directive is used in the **.h/.c** file convention described below, which is necessary to get prototypes correct.

-
  ```
  #include "foo.h"
      // refers to a "user" foo.h file
      // in the originating directory for the compile
  ```

-
  ```
  #include <foo.h>
      // refers to a "system" foo.h file
      // in the compiler's directory somewhere
  ```

#include

Thus, the preprocessor replaces

```
#include <stdio.h>
```

with the contents of the file **`stdio.h`**

This is a ***header*** file for the standard IO library.

- Includes a declaration for the printf function, called a ***function prototype***.
- Without it the call to **`printf`** would cause a compile warning.
    - Some compilers automatically include **`stdio.h`**
    - **Do not rely on that possibility!**

#include

- You could include *anything* in your program *anywhere* you like with the #include directive.

- But it is generally bad practice
  - to use **`#include`** for anything other than header files.
  - to put **`#include`** anywhere other than at the top of your program.

C Convention

- C convention:

  1. function prototypes precede function definitions
  2. clients that need to use functions, must have the function
     prototypes available ("be able to see them")

- To achieve this:

  (1) A separate file named foo.h will contain the prototypes for
  the functions in foo.c

  (2) Near the top of foo.c will be the following line

  ```
  #include "foo.h"
  ```

- This ensure that the function definitions in foo.c see the prototypes in foo.h

C Convention

- Any **`xxx.c`** file which wishes to call a function defined in foo.c must include the following line to see the prototypes

```
#include "foo.h"
```

- This ensures the "clients must see prototypes" rule above

Symbolic Constants

- We can use the preprocessor command
  `#define`
  to create meaningful symbolic names for numeric constants.

- Example:

```
#define PI 3.1416
```

- Works like a “replace” command in a word processing program.

- ***Before the actual compilation starts*** the preprocessor scans the rest of the source file and replaces each instance of **`PI`** with **`3.1416`**

Symbolic Constants

Things to notice about symbolic constants:

`PI` is not a variable!

- No memory is allocated for it.
- It disappears after the preprocessing stage

If you wrote

```
PI = 21.4;
```

the compiler would see

```
3.1416 = 21.4;
```

and flag the line as an error.

Symbolic Constants

Things to notice about symbolic constants:

There is no "=" and no ";" in a #define.

The elements are separated by spaces.

```
#define  PI  3.1416
```

space

No semicolon

Symbolic Constants

- It is traditional C programming style to use all caps for symbolic constants as was done for PI.
- The fact that a name is all caps is a ***clue*** to the *reader* that it is a symbolic constant.
  - The compiler doesn't care.
  - Don't use all caps for other names.

Warning

- The use of complex textual substitution in a preprocessor is a *flawed concept*.

- The compiler does not see what you see.
  - Mistakes can be extremely difficult to figure out.
  - Debuggers don't see the same code that you wrote.
- It was considered a cool idea in 1970.
- It permits you to create bizarre effects.
  - Other than for named constants, Don't!

Symbolic Constants

- **`#define`** is generally beneficial so long as you use it only to define names for numeric constants as in the **`PI`** example.
  - More creative uses often cause trouble.

- Later in the course you will learn how to designate a *variable* as a constant.
  - Sounds like an oxymoron!
  - Actually good practice.
  - Cannot always be used in place of **`#define`**

Curly Brackets

Curly brackets mark the beginning and end of a ***block*** of code.

```
int main()
{
   printf( "Hello, World! \n" );
   return 0;
}
```

- Every function body is a block.
  - We will see other uses of blocks in the future.

- Align the start and end brackets for each block.
  - Indent everything inside the block by three spaces.
  - Easily done in Xemacs: hit Tab key while in the line

Curly Brackets

- Align the start and end brackets for each block.
  - Indent everything inside the block by three spaces.

- This is a matter of ***programming style***.
  - Makes the program easier to read and understand.
  - No effect on what the program does.
  - Compiler doesn’t care.

- Many variations
  - Most organizations have their own style guidelines.
  - No widely accepted single standard.

Block Structure

- Most "standard" languages use curly braces to specify a block
  - C, C++, original Basic, Java, ...
- Some newer languages, like Python, use indentation to define a block

Variables

- A *variable* is a named location in the program’s memory, where it can store a value for later use, such as a number or a character, or … .

- Use descriptive names for your variables
  - Write for the reader!

- Avoid the predefined *keywords* of the C langage.
  - See page 26 in the text.

Identifiers

- Variable names are *identifiers*.
  - Letters, digits, underscores.
  - First character cannot be a digit.
    - Avoid using underscore for first char.
    - Legal, but may duplicate a name defined in a system library.
  - Case sensitive.
  - No maximum length.
    - Avoid names longer than 10 or 15 characters for readability.

Variables

- Each variable has a *type*.
  - Specifies how it is to be used.
  - Determines the amount of memory used.
- Examples of "simple" types with typical sizes

int	32 bit integer
double	64 bit floating point number
char	8 bit character (letter)

- Sizes can vary from one system to another

Types

- **float** is a 32 bit floating point type
  - 7 significant digits
  - Used by examples in textbook
  - **DO NOT USE `float` IN YOUR PROGRAMS – USE `double` INSTEAD.**
- **double** is a 64 bit floating point type
  - 15 significant digits
- Remember: Do not use float in this course
- **Use double**.

Types

- C provides *many* more types
  - Unsigned versions
  - Long ints
- For most purposes you only need
  - int
  - double
  - char
- Memorize these
  - Use for all progarmming projects in this course unless the assignment specifies otherwise.

Addresses

- Each variable has a *memory address* determined by the compiler and linker.
- We use the variable’s *name* in program statements.
- Actual machine instructions in the compiled program use *addresses*.
- The variable’s name is *bound* to its address during the process of compiling and linking (symbol table).

Variable Names and Addresses

- Most of the time* we have no reason to be concerned with a variable’s address
- If we do need the address, use & in front of the variable name
- Example:
  - int average
  - &average represents the address of the location associated with the integer variable “average.

Input and Output

- We have alredy seen one way to output a string of characters to the user’s terminal:

\n says to start a new line

```
printf ("Hello, World!\n");
```

- **`printf`** allows us to insert numbers into the output with a specified format.
  - Examples:
    - Format as integer
    - Format as 10 digit number with 4 decimal places

printf

```
int n1;
...
printf ("Here is an integer: %d \n", n1);
```

% says that this is a format specifier.
d says to format the variable as a decimal integer.

The contents of the variable specified by the **first argument** after the format string will replace the **first format** specifier in the output.

There could be additional format specifiers and additional arguments to replace them.

printf

```
double n3;
...
printf ("Here is a real number: %10.4f \n", n3);
```

Do NOT use `lf` in `printf`

f says to format the variable as a real number. (floating point)
.4 says to format the number with four decimal places.

`printf` will round if necessary.

10 says to reserve 10 character positions in the output for this number (including the decimal point.)

If fewer than 10 characters are needed for the number, spaces
are appended on the left to make the string have length 10
If more than 10 characters are needed, all needed characters
will be printed.

printf Examples

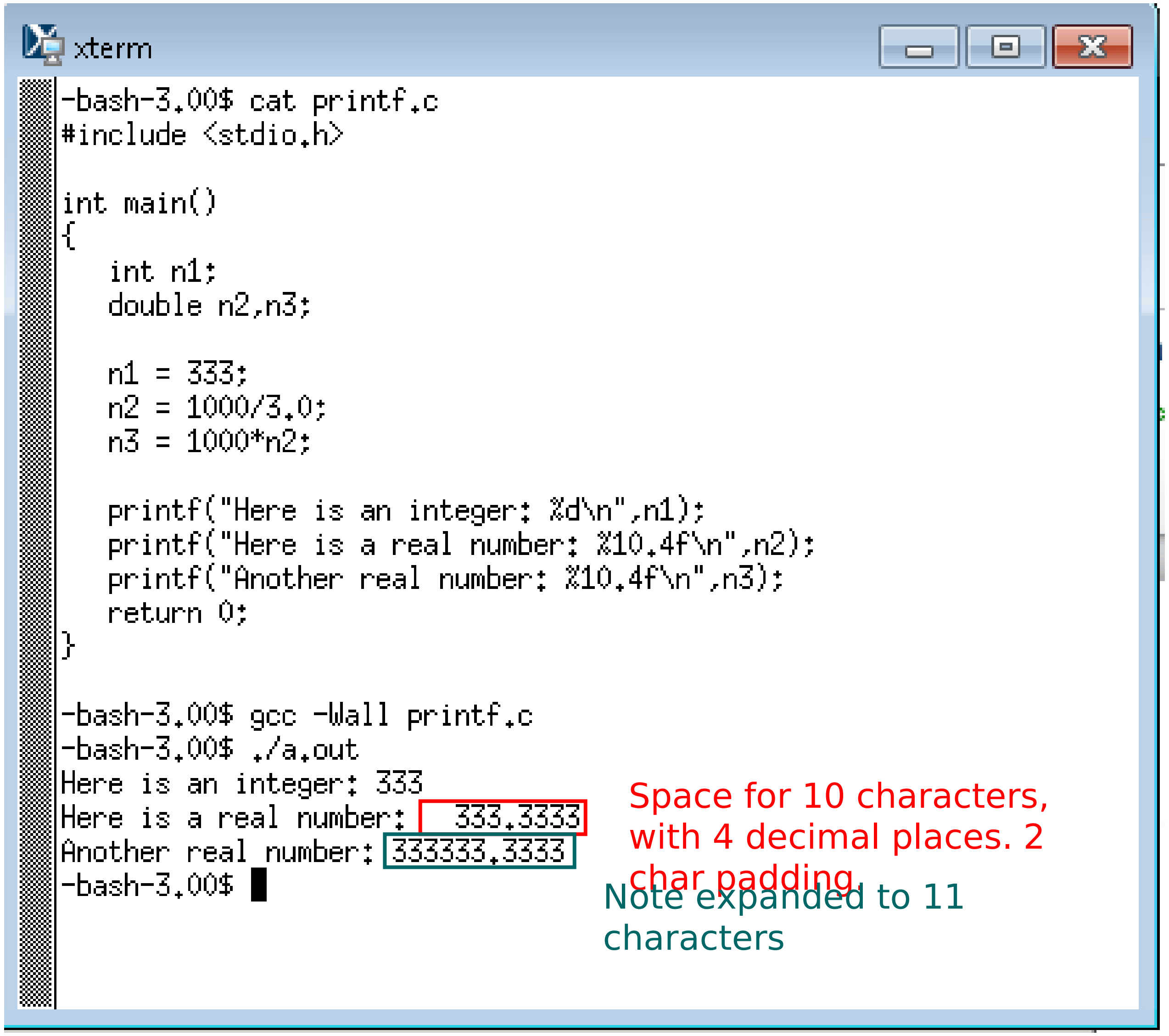
```
#include <stdio.h>

int main()
{
    int n1;
    double n2;

    n1 = 333;
    n2 = 1000.0 / 3.0;

    printf ("Here is an integer: %d \n", n1);
    printf ("Here is a real number: %10.4f \n", n2);
    return 0;
}
```

printf Examples



```
-bash-3.00$ cat printf.c
#include <stdio.h>

int main()
{
   int n1;
   double n2,n3;

   n1 = 333;
   n2 = 1000/3.0;
   n3 = 1000*n2;

   printf("Here is an integer: %d\n",n1);
   printf("Here is a real number: %10.4f\n",n2);
   printf("Another real number: %10.4f\n",n3);
   return 0;
}

-bash-3.00$ gcc -Wall printf.c
-bash-3.00$ ./a.out
Here is an integer: 333
Here is a real number:   333.3333
Another real number: 333333.3333
-bash-3.00$
```

Space for 10 characters, with 4 decimal places. 2 char padding.

Note expanded to 11 characters

printf

There are *many* additional possibilities for formatted output.

These few will do for a good while.

Learn others as you need them.

- Reference book on the C language
- Search for printf on google

Caution Regarding printf

- The format specifier *must match* the type of the variable to which it applies.
  - The compiler does not check unless you compile with ***-Wall***.
  - When you use the Compile/Validate button in the IDE, the system will add extra checking options, including ***-Wall***.
  - You should get in the habit of using the ***-Wall*** option in all your compile commands after this course!

Input Formats

- The scanf function takes an **input format** string followed by a list of ***addresses*** of variables.

- You are basically specifying the *location* to place the scanned value.

```
scanf ( "%d" , &weight);
```

- Note difference from **`printf`**
  - **`printf`** takes *names* of variables to be printed
  - **`scanf`** takes *addresses* of variables for input

- Note real number format specifiers:
  - "f" for float (which we do **NOT** use)
  - "lf" for double

Keyboard Input

- Initialize variables to 0 when they are declared or very soon afterward.
  - or a reasonable default value.
- Output a prompt to tell the user what to enter.

Keyboard Input Example

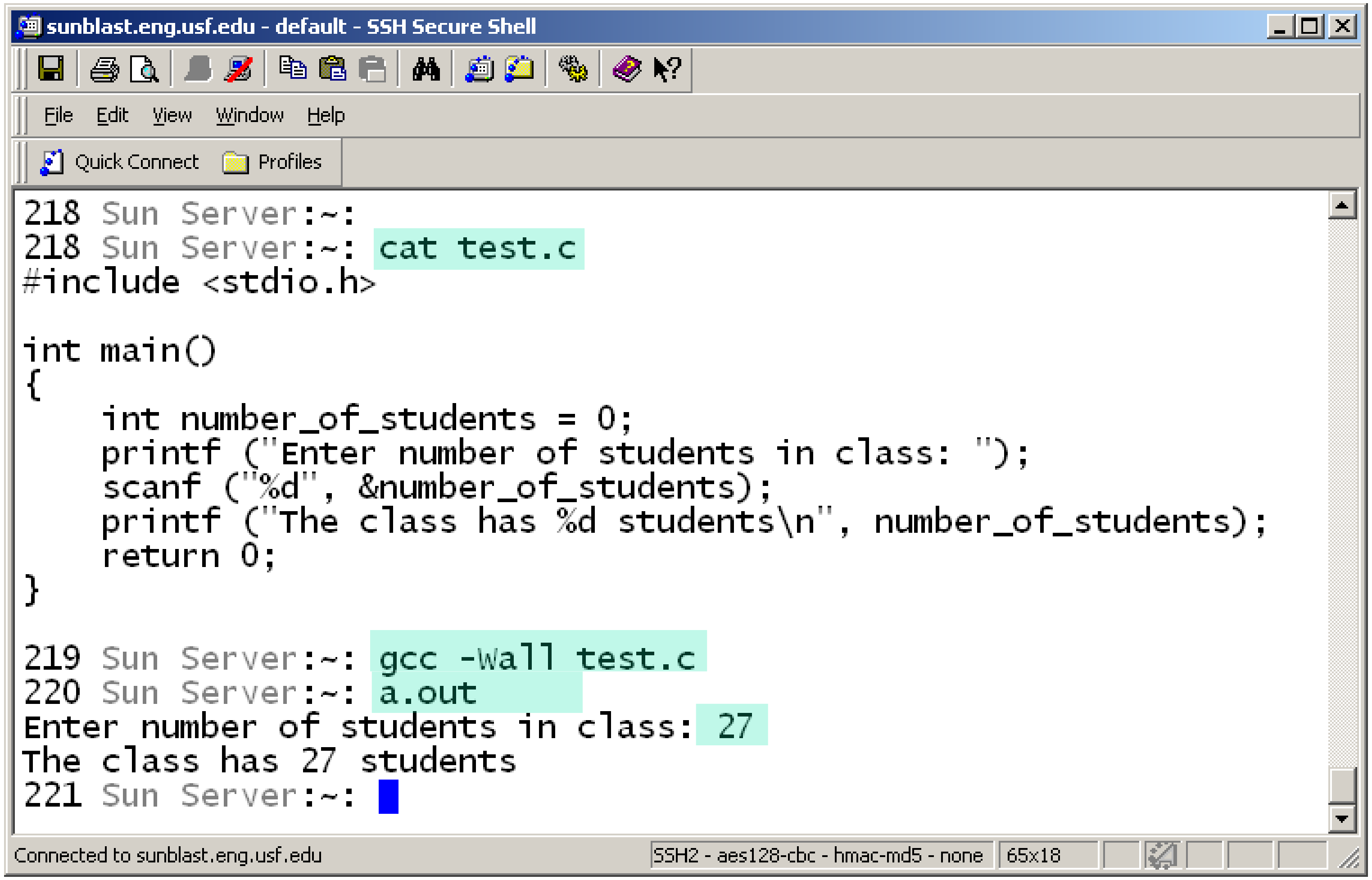
variable declaration

initializer

```
#include <stdio.h>

int main()
{
    int number_of_students = 0;
    printf ("Enter number of students in class: ");
    scanf ("%d", &number_of_students);
    printf ("The class has %d students\n", number_of_students);
    return 0;
}
```

Keyboard Input Example

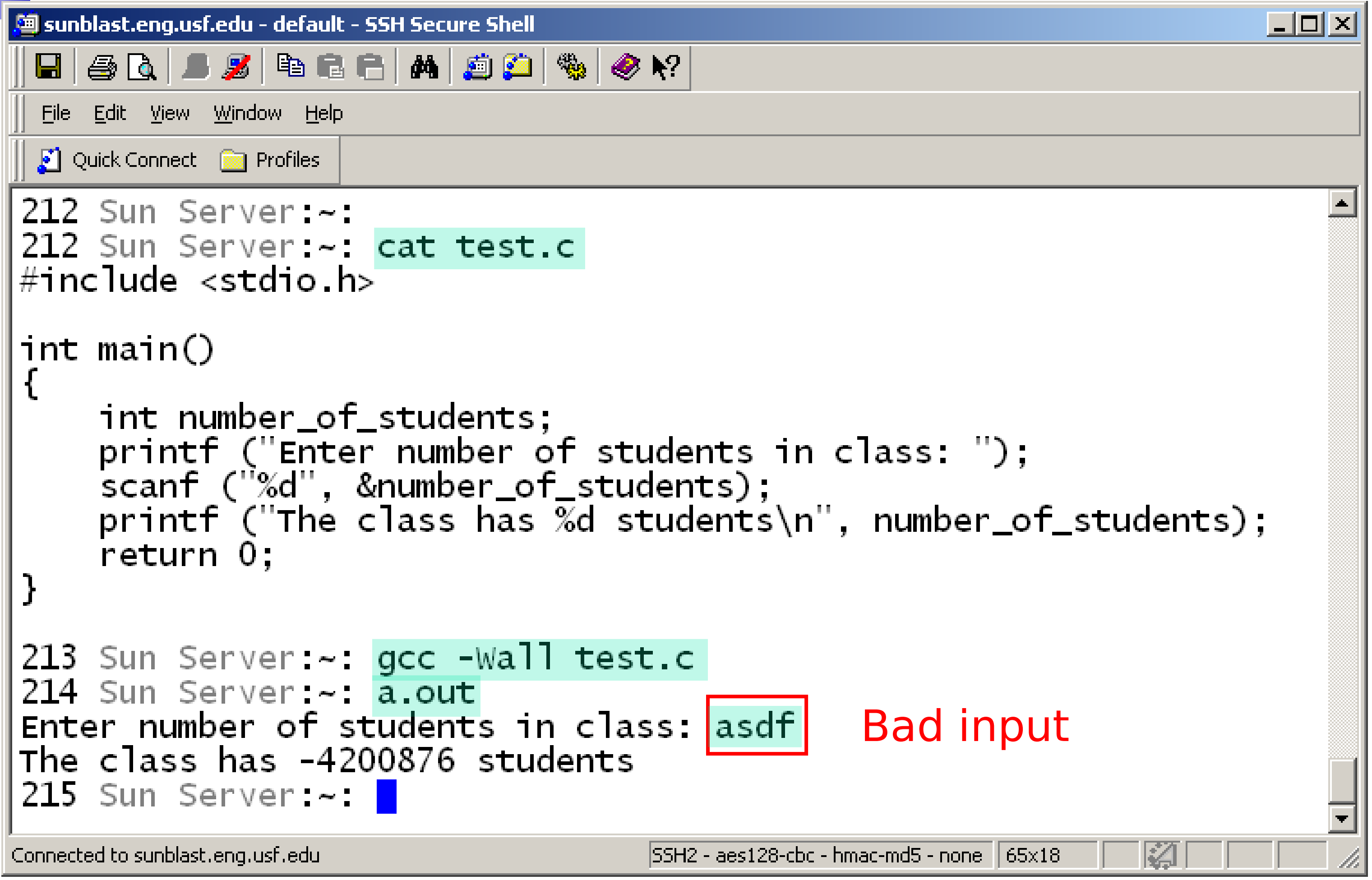

sunblast.eng.usf.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
Connected to sunblast.eng.usf.edu
SSH2 - aes128-cbc - hmac-md5 - none
65x18

```
218 Sun Server:~:
218 Sun Server:~: cat test.c
#include <stdio.h>

int main()
{
    int number_of_students = 0;
    printf ("Enter number of students in class: ");
    scanf ("%d", &number_of_students);
    printf ("The class has %d students\n", number_of_students);
    return 0;
}

219 Sun Server:~: gcc -Wall test.c
220 Sun Server:~: a.out
Enter number of students in class: 27
The class has 27 students
221 Sun Server:~:
```

Why initialize input variables?



sunblast.eng.usf.edu - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles

```
212 Sun Server:~:
212 Sun Server:~: cat test.c
#include <stdio.h>

int main()
{
    int number_of_students;
    printf ("Enter number of students in class: ");
    scanf ("%d", &number_of_students);
    printf ("The class has %d students\n", number_of_students);
    return 0;
}

213 Sun Server:~: gcc -Wall test.c
214 Sun Server:~: a.out
Enter number of students in class: asdf
The class has -4200876 students
215 Sun Server:~:
```

Connected to sunblast.eng.usf.edu
SSH2 - aes128-cbc - hmac-md5 - none
65x18

Bad input

The first character terminated the scan.

The input variable was left uninitialized.

asdf remains in the input buffer, so a second scan would also fail

Input Validation

- A real program should check that the input value is valid.
- If input is not valid, ask the user to try again.
- Thus, input validation will use a loop for each data item being input:
  - input the item
  - while input not valid:
    - print a message requesting re-input
    - input the item

Scanf() error check

Test code under course Blackboard page → Modules → Tutorial

scanf_error_check.c //Hint: Useful for Assignment1

clear_input_buffer.c //Hint: Useful for Assignment2

Assignment Statements (Review)

- We can set the value of a variable using the = operator.

```
        int minutes;
        ...
        minutes = 10;
```

- This means “Set minutes equal to 10”
- Notice data moves from right to left.
- Read as “minutes gets 10.”
- Called an “***assignment statement***.”

We are “assigning” the value 10 to the variable minutes.

Assignment Statements (Review)

```
int minutes;
...
minutes = 10;
```

Right hand side (RHS) can be a literal value as above

Or it can be a variable

```
minutes = time_to_go;
```

Or it can be the result of a calculation.

```
minutes = seconds / 60;
```

Assignment Statements (Review)

- Left hand side *must* be a variable.

  When we write

  ```
          var1 = var2;
  ```

  we are saying

  “Replace the contents of variable **`var1`** with the contents of variable **`var2`**.”

The previous contents of var1 are lost forever.

The contents of var2 are not affected.

What You Should Do Now

Go to the course page on Blackboard → Modules → Tutorial

Go through the tutorial and setup your Circe account

Compile and run the test code provided

If you have any troubles come see me as soon as possible. You will be using your Circe account to test your assignments

What You Should Do Now

- Read Chapter 2
- Look at the Q & A section
  - Be sure you understand
- Look at the Exercises
  - Try some for yourself.
  - Note that solutions to exercises marked with ● W are available on the author's web site.

    (http://www.knking.com/books/c2/answers/index.html)
- Look at the Programming Projects.
  - Think about how you would do them.
  - Try some for yourself.
  - Again, solutions are available on the author's website for those projects marked with ● W
- If anything doesn't make sense, ask for help!