PROGRAMMING IN C

STYLE

- Code style is something students do not like.
- It is often left until AFTER the program is finished as an extra added chore.
- Style is about being sensitive to how easy it is for a co-worker to read and update your code.
- To most organizations, if you write the best code possible, but no one can read it or understand it, it is not worth anything.
- In many cases, badly formatted, hard to read, hard to understand code is worse because instead of modifying your great code when needed, they end up having to throw it out and restart.
- I personally feel style is AS IMPORTANT as the code itself.
- And after working with others and having to modify other peoples code, you will eventually come to that same conclusion.

USING STYLE

- Style should not be an afterthought.
- Do it as you type.
  - Put spaces between things.
  - Put braces in the right place.
  - Tab as you go along.
  - Name things correctly according to convention
- You will become used to it. It will become second nature. Once you get used to it, you will find it doesn't take any more time.
- If you use a style checker, you will still need to make a few modifications at the end, but they will be minor.

WHY USE STYLE

- It is professional.
- Employers usually have large code bases. What matters most to them is maintainability. Sloppy code is not maintainable code.
- Sloppy coders are lazy coders. You can do both with a minimum of effort. The fact that you choose not to be neat as you type says something about your work ethic.
- It makes troubleshooting much, much easier.
  - Finding a missing bracket in sloppy code is a guessing game.
  - There may be multiple places you can put a bracket that will satisfy the compiler but be logically incorrect.
  - If you have ever played the "maybe the bracket goes here" game, that is the sign of a noob programmer. If you're not a noob, then stop acting like one.
  - Properly indent and where brackets go isn't ever a guessing game. You either simply will not make the error, or it will be very, very obvious where the bracket goes when you do forget one.

STYLE IN THIS CLASS

- I don't care too much about which conventions get used.
- But I do expect you to select one and stick with it.
- You MUST use indention on every single one of these programs.
  - I will count off on your grade if you don't indent correctly and consistently.
- Brackets must be used consistently, or you will lose points.
  - If you like brackets on the same line as a statement, then do that always.
  - If you like brackets on their own line, then do that consistently.
- You should not write single huge functions. Break functions into logical pieces and move those to other functions and call them.
- Comments are extremely important. But for the purposes of testing, I wont generally require them since the time on the test is limited.

BREAK AND CONTINUE AND MULTIPLE RETURNS

- These have their place, but they should be used extremely sparingly inside of a loop. (Break in a switch is fine.)
- You should always arrange your loops in such a way as to remove break and continue.
- The code on the left obscures the fact that code1, code2, and code3 only get run if x >= 0.
- The code on the right makes it more clear through indention when code1, code2, and code3 get executed.
- Multiple returns have the same effect.

```
int x = 0;
while (x > 0) {
        scanf("%d", &x);
        if ( x < 0) break;
        code1;
        code2 ;
        code3;
}
```

```
int x = 0;
while (x > 0) {
        scanf("%d", &x);
        if ( x >= 0) {
                code1;
                code2;
                code3;
        }
}
```

MORE OBSCURITY

- What will the following print?
- If you look at the for-loop control, you would think it runs 10 times.
- It is easy to see the break in the code below, but the more code there is inside the for-loop, the less easy it will be to pick out the break.
- You shouldn't have to scour the code to see how many times the for-loop runs. The point of a for-loop is to do something a specific number of times.

```
int z = 5;
int y = 1;
for (int x = 0; x < 10; x++) {
        printf("%d ", x);
        if (x + y < z) break;
}
```

MORE OBSCURITY BETTER

- What will the following print?

```
int z = 5;
int y = 1;
for (int x = 0; x <= (z-y); x++) {
        printf("%d ", x);
}
```

USING GOTO

- Don’t use goto.
- There **ARE** some very, very specific places where goto is a good solution. But those specific places almost never show up in regular programming.
- If you think goto is necessary in your program, it almost always means you have bad logic in your code.
- My suggestion is to forget goto even exists.
- The Web-CAT tests will fail if you use goto.

GOTO, CONTINUE, BREAK, MULTIPLE RETURNS

- The issue with these instructions is that in most cases they obscure code that could have been written more cleanly.
- Throwing in a break when you are done with a loop might make you get through a little quicker, but when someone comes behind you to update that code it will make it slower for them.
- Businesses want good code, but they also want easily modifiable code. If you show up to an interview with lots of breaks and continues in your example code or you use those to answer interview questions, when a simple if would have been clearer and just as effective will reflect negatively on you.
- Break, continue, multiple returns, and even goto have their place, BUT you will have cleaner, more logical code without them and their use should be minimized as much as possible. In this class it is always possible, and Web-CAT will fail if you use them.
- So, just to give you more practice at eliminating these items that cause a reduction in code clarity, for the C programs in this class you will not be allowed to use them.

HELLO WORLD

```
#include <stdio.h>

int main(void)
{
		printf("Hello world!!\n");
}
```

COMPILING ON STUDENT

- What do you need to run "Hello World"?
  - Access to computer with a compiler
- Computer
  - student.cs.appstate.edu
  - connect with putty
- Compiler
  - GCC The Gnu Compiler Collection

ALTERNATIVES TO USING STUDENT

- ASNI C is a standard version of C. Any compiler which adheres to ANSI C should give the same output.
- Any LINUX or Mac can install GCC/G++.
  - Most implementations of LINUX have it already installed
- Windows has many C compilers.
  - Visual C++ will compile C code.
  - mingw is an excellent command line based, windows implementation of gcc or g++.
  - Dev-C++ is a good development environment.
  - Use bash on Ubuntu on Windows (See additional slide).
- There are various web pages that let you enter code, compile, and run right on the web page.
- Not being able to get your own computer to compile programs will not be an excuse for not getting assignments completed. I expect you to at least be able to use student; make sure you can.

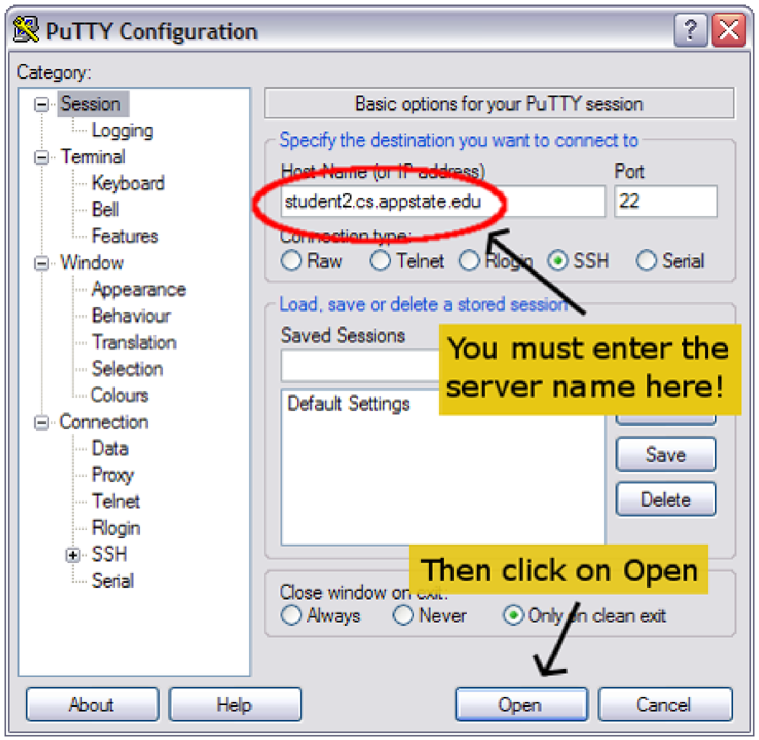
CONNECTING TO STUDENT

- PuTTY is a Windows program for connecting to servers running the secure shell protocol (ssh).
  - http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html
  - Download putty.exe and save it to your desktop. You don't have to install anything, just click on it.
- LINUX machines and Macs, open a terminal or shell window and use the ssh command
  - Type the following

```
ssh    your_user_name@student2.cs.appstate.edu
```

USING PUTTY

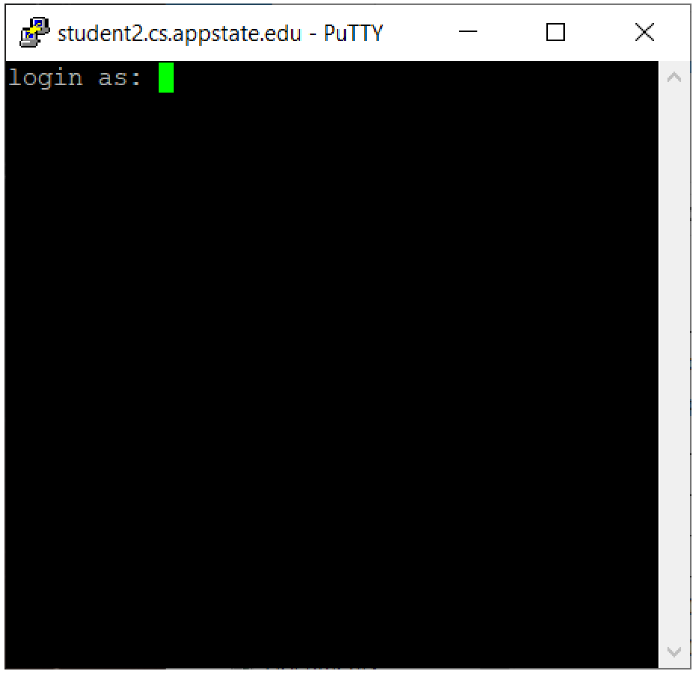
- There are many options you can set on putty.



Host Name (or IP address): student2.cs.appstate.edu
Port: 22
Connection type: Raw Telnet Rlogin [✓ SSH] Serial
Close window on exit: Always Never [✓ Only on clean exit]

GETTING LOGGED IN

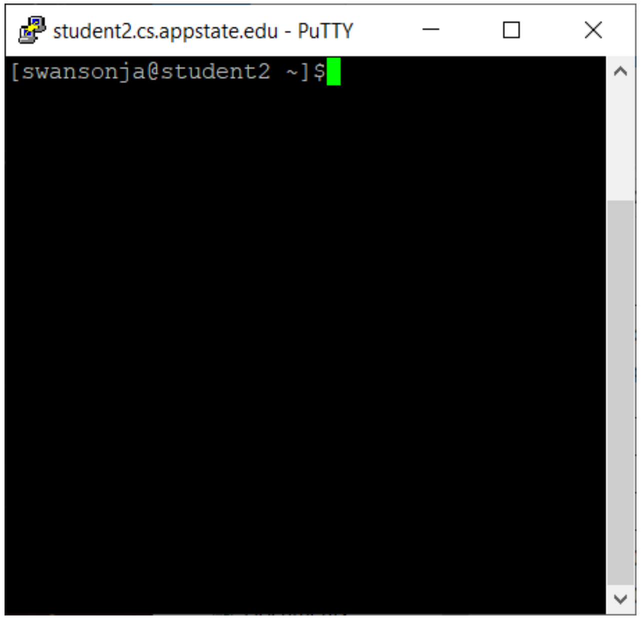
- Use your username exactly as you use on other systems at ASU.
- Your password will be your BANNER ID if you have never logged into student.
- If you have logged into student you will have to remember your old password. Your AsULearn password will almost definitely not work.
- Contact the system administrator if you cannot remember your password.
  - https://compsci.appstate.edu/chg-password
- When you are typing your password, note that no characters show up. This is normal on UNIX system.



```
login as:
```

LOGGED IN

- If you see a window similar to this you are logged in.
- The text on the screen is a command prompt and it is waiting for a command to be entered.



```
[swansonja@student2 ~]$
```

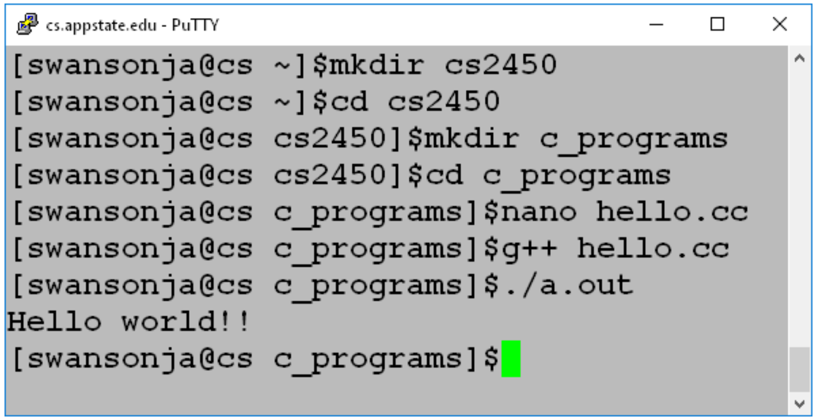
SOME UNIX COMMANDS

- ***pwd*** shows you the current folder you are in. The present working directory (pwd).
- ***ls*** list all files in the pwd.
- ***mkdir*** create a new directory or folder in the pwd.
- ***cd*** change to a different folder. That folder must be in the pwd.
- ***cd ~*** change back to your home directory.
- ***rm filename*** removes a file named *filename* from the pwd.
- ***rmdir directoryname*** removes an EMPTY directory.
- ***rm dirname –rf*** removed directories and ***EVERYTHING*** in that directory
- ***mv oldfilename newfilename*** - Rename a file.
- ***ctrl – c*** will kill an out of control process (i.e. infinite loop).
- ***exit*** – exits the terminal. Quit putty this way, don't use x at top of window. Always exit your editor and exit putty using ***exit***.
- ***man*** – Get help (manual pages) on a command.
- For example, man g++ will show you all the g++ command line options.

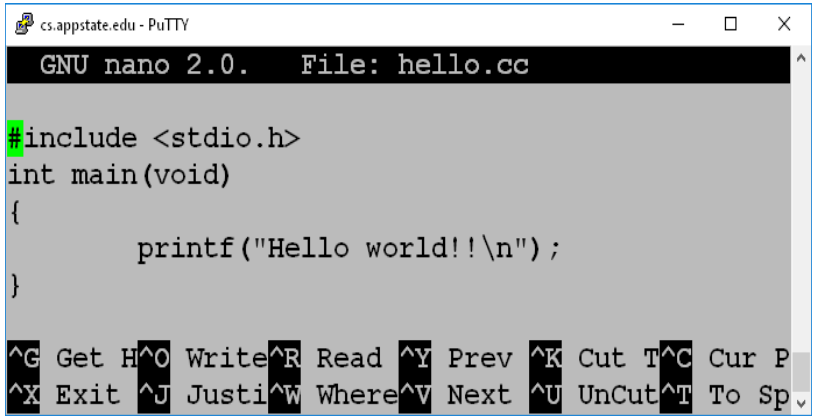
EXAMPLE HELLO WORLD

- Log into student.
- Make a new directory for 2450.
- Change into that directory.
- Make a new directory to store **C** programs.
- Change into that directory.
- Open an editor and type in the hello world program from slide 2.
- Save the program as hello.cc and exit the editor.
- Compile the program.
- Run the program.

EXAMPLE IMPLEMENTED



```
[swansonja@cs ~]$mkdir cs2450
[swansonja@cs ~]$cd cs2450
[swansonja@cs cs2450]$mkdir c_programs
[swansonja@cs cs2450]$cd c_programs
[swansonja@cs c_programs]$nano hello.cc
[swansonja@cs c_programs]$g++ hello.cc
[swansonja@cs c_programs]$./a.out
Hello world!!
[swansonja@cs c_programs]$
```



```
  GNU nano 2.0.    File: hello.cc

#include <stdio.h>
int main(void)
{
        printf("Hello world!!\n");
}

^G Get H^O Write^R Read ^Y Prev ^K Cut T^C Cur P
^X Exit ^J Justi^W Where^V Next ^U UnCut^T To Sp
```

- I used nano as an editor. Feel free to use vi/vim, emacs, or any other text editor you are comfortable with.
- Control-O (WriteOut) saves from nano. Make sure it is named hello.cc.
- Control-X exits.
- The default name of the executable file created by gcc is a.out.
- You can change the output file name by using the –o option
- The above uses gcc to compile. gcc and g++ do the same thing.

COMPILING A PROGRAM

g++ hello.cc

Creates an executablefile named a.out

Run by typing ***a.out*** or ***./a.out*** on the command line

g++ -o hello hello.cc

Creates an executable file named hello.

Run by typing ***hello*** or ***./hello*** on the command line

g++ -Werror -Wall -o hello hello.cc

Creates an executable file named hello and treats all warnings as errors.

-Werror : Treat warnings as error

-Wall : Show all warnings

-o : Name the output file

(-**Werror WILL BE TURNED ON FOR WEB-CAT**).

WARNINGS AND ERRORS

```
#include <stdio.h>
int main()
{
	double a;
	printf("Enter a double: ");
	scanf("%f", &a);  //error should be %lf
	printf("%f\n", a);
}
```

The above will give a warning and compile.

Typing 1.2 at the prompt will result in 0.00000 as output.

-Werror will make sure this doesn’t happen.

ASSIGNMENTS

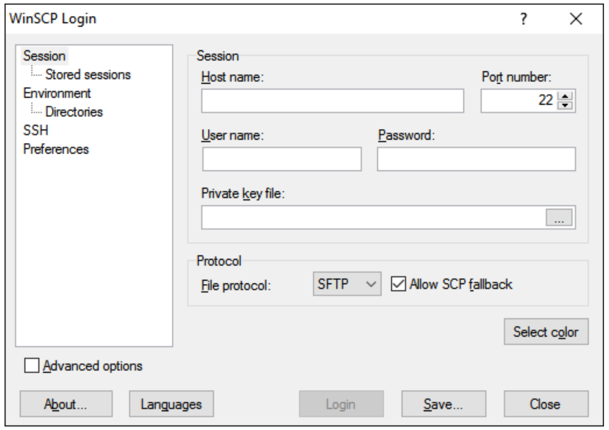
- Assignments will be broken into two parts.
  - You will create a file named main.cc with a main method in it. Sometimes I will give you this file.
  - You will create separate file with your program code. The program code will be a series of functions. **DO NOT PUT A MAIN IN YOUR CODE FILE.**
  - Web-CAT uses a main method to test your code. If your code already has a main method it will fail.
  - Note that we are learning C even though we are using C++ to compile.
    - For this class you are NOT allowed to use the C++ language.
    - Most specifically you are not allowed to use CIN or COUT.
    - You are only allowed to use code as shown in my slides or the book.

SUBMITTING TO WEB-CAT

- Submitting
  - Go to the Web-CAT web page
    - http://webcat.cs.appstate.edu
  - Select the assignment
  - Upload
- Grading
  - You will be graded according to the number of tests that you pass.
  - You will lose points for excessive submissions
  - You will lose points for every day the assignment is late.
  - You will receive a 0 if the assignment is more than 2 days late.

GETTING FILES FROM STUDENT

- Windows users can use WinSCP.
  - Host name: student.cs.appstate.edu
  - Enter your user name and password.
    - Note that your password is your password on the student server and NOT your AsULearn password.



Session

Host name:

Port number: 22

User name:

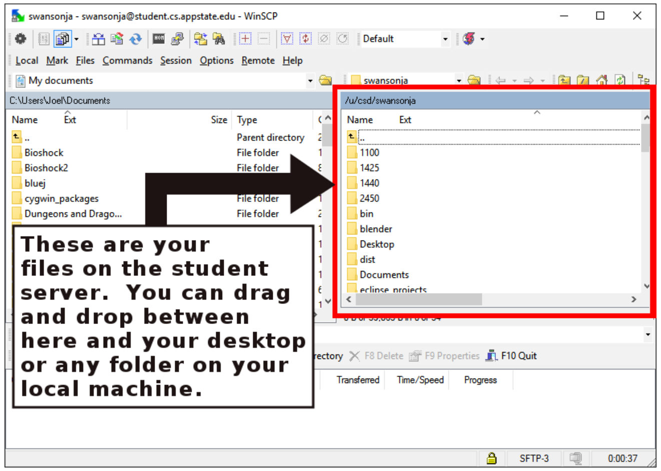
Password:

Private key file:

Protocol

File protocol: SFTP

☑ Allow SCP fallback


swansonja - swansonja@student.cs.appstate.edu - WinSCP
Default
Local Mark Files Commands Session Options Remote Help
My documents
swansonja
C:\Users\Joel\Documents
/u/csd/swansonja
Name Ext Size Type
Name Ext
F8 Delete F9 Properties F10 Quit
Transferred Time/Speed Progress
SFTP-3
0:00:37

Name	Type
..	Parent directory
Bioshock	File folder
Bioshock2	File folder
bluej	
cygwin_packages	File folder
Dungeons and Drago...	File folder

Name
..
1100
1425
1440
2450
bin
blender
Desktop
dist
Documents
eclipse_projects

These are your files on the student server. You can drag and drop between here and your desktop or any folder on your local machine.

LINUX AND MAC USE SCP

- Assuming
  - I have a folder named 2450 with a folder named assign1 inside on student.
  - My PWD is my assign1 directory on the local machine

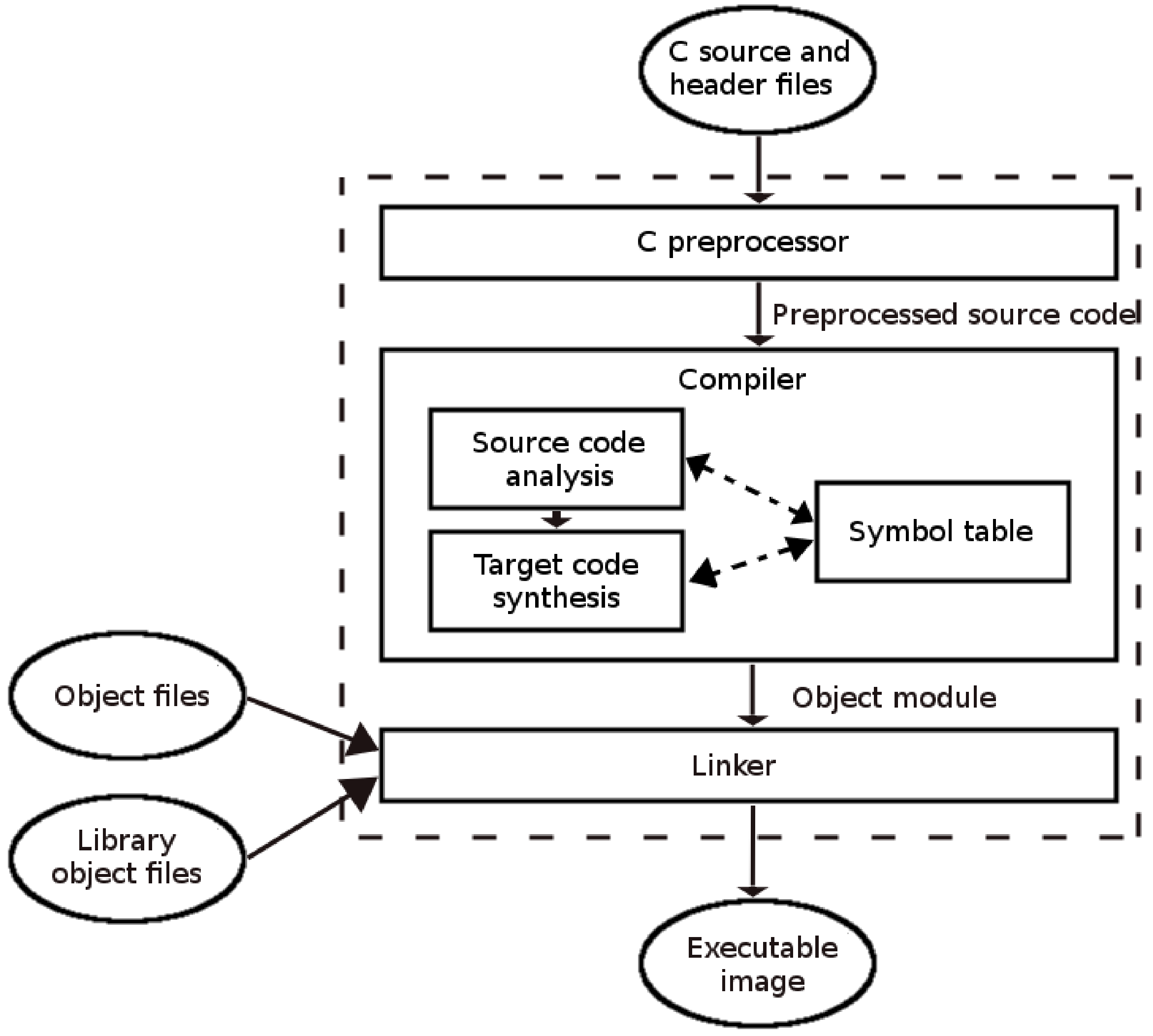
- Copy file TO student FROM the local machine.

```
scp   assign1.cc
swansonja@student2.cs.appstate.edu:2450/assign1/assign1.cc
```

- Copy assign1.cc FROM student TO the local machine

```
scp
swansonja@student2.appstate.edu:2450/assign1/assign1.cc
assign1.cc
```

COMPILING DETAILS



PART OF THE COMPILER

- Preprocessor – Processes directives
  - Directive start with the # symbol.
    - #include will copy the named file into the source code buffer.
    - #define will replaced any instance of the constant with the value specified in the entire source code buffer.
  - Output is preprocessed source code.
- Compiler – Convert code to assembly language and track variable names in the symbol table. This portion will attempt to optimize code. The symbol table is a mapping of the variable names to memory locations actually used.
  - Output object code – Not all addresses resolved.
- Linker – Determines and inserts the addresses for any external code that is needed such as code for building and displaying windows.
  - Output is executable code ready to be executed by the hardware.
- GCC seems to do all of the above in one step, but it actually performs each step separately.

C AND JAVA

- I am assuming you know Java.
- Java and C are similar in syntax.
- Methods are called functions but are otherwise declared and used the same. There is no public or private, however.
- The "***main***" function is where code operation begins.
- ***if*** and ***while*** work the same.
- Operators all act the same.
- ***for*** loops act the same (In older versions of C you must declare loop control variables before the loop).

HOMEWORK 6 – LOOPS AND SHAPES

- Since most of the statements and concepts from this first assignment are almost exactly the same as Java you should be able to figure it out.
- If not, ask.

VARIABLES

- Three basic primitive types – work the same as java
  - int
  - char
  - double
- No String as a type.
- Local variables – defined in methods.
- Global variables – defined outside of methods. Similar to fields in that they are available to all functions. Global variables, however, are available to the entire program, like a public field, and should be used sparingly.
- ALWAYS use local variables and parameter passing where possible instead of global variables.
- ALWAYS initialize your variables. Local variables are NOT initialized to zero in C.

INTEGER TYPES

Type	Size	Range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615
long long	8 or 16	Long longs can be the same as long or
unsigned long long	8 or 16	will be twice as big.

FLOATING POINT TYPES

Type	Size	Range	Decimal Places
float	4 byte	1.2E-38 to 3.4E+38	6
double	8 byte	2.3E-308 to 1.7E+308	15
long double	16 byte	3.4E-4932 to 1.1E+4932	19

```
#include <stdio.h>
int main()
{
        char b = 1;
        printf("%d\n", b); //Prints 1
        b = -1;
        printf("%d\n", b); //Prints -1
        b = 128;
        printf("%d\n", b);  //Prints -128
        b = 0xff;
        printf("%d\n", b);  //Prints -1
        b = 0xf0;
        printf("%d\n", b);  //Prints ____?
        b = 165;
        printf("%d\n", b);  //Prints -91
        b = 1.5;
        printf("%d\n", b);  //Prints 1
        b = 'c';
        printf("%d\n", b);  //Prints 99
}
```

DECLARING VARIABLES

- The same as Java

```
double width;
double pType = 9.44;
double mass = 6.34E2;
double verySmall = 0.1094E-31;
int average = 12;
int windChill = -21;
int unknownValue;
char initial = 'A';
char num4 = '4';
```

DECLARING CONSTANTS AS #DEFINE

- Constants

```
#define LETTER      '1'
#define ZERO                    0
#define NUMBER      123
```

- Constants do not have a type.
- The text will simply replace the constant identifier in the source code during the preprocessor phase before compiling begins.

OPERATORS SAME AS JAVA

- The following are the same as Java.
  - Operators and Expressions. (+, -, *, /, %, a + b)
  - Assignment (x = a + b)
  - Parenthesis ((a+b)*x)
  - Increment and decrement (++, --)
  - Combined operators (+=, -=, *=, etc…)

ORDER OF OPERATION

Precedence	Associativity	Operators
1	L to R	() (function call) [ ] (array index) ->
2	R to L	++ -- (postfix versions)
3	R to L	++ -- (prefix versions)
4	R to L	*(indirection) &(address of) +(unary) –(unary) ~ ! sizeof
5	R to L	(type) (type cast)
6	L to R	* (multiplication) / %
7	L to R	+ (addition) - (subtraction)
8	L to R	<< >>
9	L to R	< > <= >=
10	L to R	== !=
11	L to R	&
12	L to R	^
13	L to R	|
14	L to R	&&
15	L to R	| |
16	L to R	? : (conditional expression)
17	R to L	= += -= *= /= %= &= |= ^= <<= >>=

RELATIONAL AND CONDITIONAL OPERATORS

- **DIFFERENT THAN JAVA**
- **C** has no Boolean data type.
- Technically
  - 0 is false
  - anything not zero is true

3 == 5 evaluates to 0

3 != 5 evaluates to 1

TRICKY ERROR

- The following code will print the word same.
- In java it would have produced an error.
- Be careful about accidentally using = instead of ==

```
int x = 5;
int y = 7;
if ( x = y )
{
        printf("Same");
}
```

TRICKY LOGICAL OPERATORS

```
int f = 7;
int g = 8;
int h;
h = f & g;
h = f && g;
h = f | g;
h = f || g;
h = ~f | ~g
h = !f && !g;
h = 29 || - 52;
```

IF AND WHILE

- The following work the same as Java.
  - if
  - if-else
  - if-else-if
  - if-else-if-else
  - while
  - do-while

PRINTF

- No print or println, only printf
- Similar to Java printf
- %[flags][width][.precision][length]**specifier**
- **Specifiers** (some of them)

  c – character (Display the character for an ASCII value)
  d or i – Signed integer
  e or E – Scientific notation
  f – floating point
  s – string of characters (null terminated)
  u – unsigned integer
  x or X – unsigned hexadecimal (lower or upper case)
  p – pointer address
  n - nothing

PRINTF FLAGS AND WIDTH

- %[**flags**][**width**][.precision][length]specifier
- Flags

-	Right justify
+	Include plus sign for positive numbers
0	Left pad number with zeros. Only for numbers with width specified.

- Width Minimum number of spaces

  * specify width as an additional integer value argument

PRINTF PRECISION AND LENGTH

- %[flags][width][**.precision**][length]specifier
- .precision

    .num The number of decimal points to display.

    .* The number of decimal points is specified as an additional integer value argument

- length used for different sizes of integers or doubles (i.e. long or short)

    h, l, or L

FUNCTIONS

- Functions follow the same rules as Java.
- No public or private though.
- Return types the same.
  - Use void type if there is no return.
  - Use return statement to return values and exit the function.
- Passing parameters is the same except
  - Pointers allow call by reference for primitive types.
  - More on this later.
- Creating functions

```
int get2Times (int x) {
        return x * 2;
}

void print2Time(int x) {
        printf("%dx2=%d\n", x,  x * 2);
}
```

- Using functions

```
        int y = get2Times(5);
        print2Times(4);  //Prints 4x2=8
```

POINTERS

- In Java, object variables reference the object they are storing.
- The variable holds the memory location or reference of the position in memory where the object was created.
- In C, we can have references to any variable.
- Using pointers incorrectly can have unpredictable results and cause hard to diagnose errors.
- Java does not allow user created reference variables.

POINTER OPERATIONS

- Declare a variable named ptr which is a pointer to an integer.

  * - dereference - Read as "star" or "splat"

  & - "address of" operator

```
int count = 5;
int *ptr;
ptr = &count;
*ptr = *ptr + 1;
printf("%i", count);
```

DEREFERENCE (*) AND ADDRESS OF (&)

- * has two uses
  - Declare a pointer variable.

    ```
    int *ptr;
    ```
  - Dereference a pointer variable.

    ```
    *ptr = 5;
    ```
  - **ONLY USE * ON POINTER VARIABLES**
- & is the "address of" operator.
  - We NEVER want the address of a pointer. So & SHOULD NOT be used on pointer variables.
  - & should be used on non pointers to get an address to assign to a pointer.

    ptr = &y; (ptr **SHOULD** be a pointer. y **SHOULD NOT** be a pointer.
- Note that
  - *ptr is used to modify the thing that ptr is pointing at

    ptr=&y;

    ptr DOES NOT have * on it (i.e., this is wrong *ptr = &y;).

    We want the address of y IN ptr.

    We do not want the address of y in the thing ptr is pointing at.

GETTING AN ADDRESS

& returns the address of a variable.

```
int count = 5;
```

&count - returns the address that C has allocated to the variable count.

POINTER EXAMPLE

```
int count1 = 1;
int count2 = 3;
int count3 = 5;
int *ptr;

int main()          {
   ptr = &count1;
   printf("%i - %i - %i\n", count1, count2, count3);

   for (int i = 0; i < 3; i++)  {
      *ptr = *ptr + 1;    // could use  (*ptr)++;
      printf("%d - %d - %d\n", count1, count2, count3);
      ptr = ptr + 1;
   }
}
```

SCANF

- Input – Similar to printf with formats
- %[*][width][modifiers]type
- The variable must be a reference or pointer.

```
int x;
scanf("%d", &x);
```

SCANF FORMAT

- %[*][width][modifiers]type
- Formats should be simple.
- Don't print prompts using the scanf format.

PROMPT LIKE THIS THIS:

```
int x = 0;
printf("Enter an integer: ");
scanf("%d", &x);
```

DON'T PROMPT LIKE THIS:

```
int x = 0;
scanf("Enter an integer: %d", &x);
```

SCANF AND SPACES

- scanf uses any whitespace character (tabs, spaces, newlines) as a delimiter.
- That means scanf will not read past a space unless you tell it to.
- The following will read 99 characters into test and will not stop until a newline (enter) is reached.

```
char test[100];
char c = ' ';
scanf("%99[^\n]", test);            or            scanf("%99[^\n]s", test);
Scanf("%c", c);  //Get rid of newline for next scanf if needed.
```

USE MULTIPLE SCANF

Good advice is to keep your scanfs as simple as possible.

Get multiple variables with multiple scanfs

```
int x, y;
printf("Enter an integer: ");
scanf("%d", &x);
printf("Enter another integer: ");
scanf("%d", &y);
```

Don't do this:

```
int x, y;
printf("Enter two integers: ");
scanf("%d %d", &x, &y);
```

SCANF EXAMPLE

```
int i = 0;
float f = 0;
double d = 0.0;
char c = ' ';

printf("Enter an integer: ");
scanf("%d", &i);

printf("Enter a float: ");
scanf("%f", &f);

printf("Enter a double: ");
scanf("%lf", &d); //Note lf (long float)

printf("Enter a character: ");
scanf("%c", &c);
scanf("%c", &c); //Why two scanfs?

printf("%i - %f - %f - %c\n", i, f, d, c);
```

ANOTHER SCANF EXAMPLE

```
int num1 = 0;
double num2 = 0;

printf("I will sum two numbers for you.\n");
printf("Enter an integer: ");
scanf("%d", &num1);

printf("Enter a double: ");
scanf("%lf", &num2);  //note the lf (long float for doubles)

printf("%d + %.2f = %.2f\n", num1, num2, num1 + num2 );

return 0;
```

WARNINGS AND ERRORS

```
#include <stdio.h>
int main()
{
	double a;
	printf("Enter a double: ");
	scanf("%f", &a);  //error should be %lf
	printf("%f\n", a);
}
```

The above will give a warning and compile.

Typing 1.2 at the prompt will result in 0.00000 as output.

ARRAYS

- Logically you can think of arrays in **C** the same way as you think of them in Java.
- The reality is that arrays are objects in Java and simply a large group of declared variables in C.

```
int grid[4];
double scores[2];
char name[4];
```

- In the above examples grid, scores, and name are simply pointers to the first element in the array
- Unlike pointers, however, grid, scores and name cannot be reassigned.
- Array size MUST be known at compile time. You cannot use an undefined variable to create an array.
- You can initialize arrays using {} (Bracket initialize)

```
int grid[]  = {1,2,3,6, 4, -3};
double scores[] = {67.0, 88.5, 21.7, 100.0, 99.2};


//This would be weird.  No null termination unless you put it in.
char name[] = {'H', 'e', 'l', 'l', 'o', '\0'};  //DON'T DO THIS!
//Just use quotes:    char name[] = "Hello"
```

ARRAY INDEXING

- [] is an operator like multiplication or addition.
- grid[index] means calculate an address like this:
- (address of grid) + index * (sizeof the type of grid)

- grid[3] = 5;
- 5 in 32 bits is ***00000000*** 00000000 00000000 00000101
- Address = 1000 * 4 * 3 = 100C
- Put ***00000000*** in 100C
- Put 00000000 in 100D
- Put 00000000 in 100E
- Put 00000101 in 100F

ARRAY LENGTH

- There is no way to determine the number of items in an array once it is created so it is vitally important to keep track.
- A good practice is to use a constant for the number of items you wish to store.

```
#include <stdio.h>
#define NUM_GRADES 10
int main()
{
	int grades[NUM_GRADES];
	for (int i=0; i < NUM_GRADES; i++)
	{
		printf("Enter grade #%2d: ", i+1);
		scanf("%d", &grades[i]);
	}
	double avg = getIntArrayAverage(grades,  NUM_GRADES);
	printf("Average: %.2f\n", avg);
}
```

GETINTARRAYAVERAGE

```
double getIntArrayAverage(int grades[], int num)
{









}
//You could use NUM_GRADES here, but don't.
//For my assignments, always pass in the number of items
//and use the parameter.  Don't assume I will use the same constant.
```

PASSING ARRAYS AS PARAMETERS

- Note the important pieces of the last two slides.

- In main:

```
int grades[NUM_GRADES];
//fill in the grades with some code of course
double avg = getIntArrayAverage(grades, NUM_GRADES);
```

- In getIntArrayAverage:

-
```
//Note the brackets [ ] are left empty.
double getIntArrayAverage(int grades[], int num)
          or
//Variable grades here is actually an int pointer.
double getIntArrayAverage(int *grades, int num)
```

MULTI-DIMENSIONAL ARRAYS

- Just like in Java we can have multi dimensional arrays.

```
int grid[3][4] ; //This is really an array of arrays
```

- You can use the bracket initializer BUT you MUST specify the array dimensions for all but first dimension.

```
int arr[][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
```

ACCESSING MULTI DIMENSIONAL ARRAY DATA

- The following will print 1 2 3 4 5 6 7 8 9 on a single line

```
int arr[][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
     printf("%d ", arr[i][j]);
    }
 }
```

PASSING MULTI-DIMENSIONAL ARRAYS

- You still need to know the sizes
- You need the correct brackets in the parameter.
- You MUST specify a dimension for all but first bracket.

```
int arr[][3] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
printArray(arr, 3, 3);


void printArray(int **test, int x, int y) {
    for (int i = 0; i < x; i++) {
       for (int j = 0; j < y; j++) {
         printf("%d ", test[i][j]);
       }
    }
}
```

DO NOT RETURN ARRAYS FROM FUNCTIONS

- **VERY BAD CODE!!!**
- **arr is a local variable and will be deallocated.**
- **Use a static array (gets created on the heap)**
- **Use a dynamic array (malloc or calloc / preferred)**

```
int *getArray()
{
    int arr[5];   //static in arr[5]
    printf("Enter the elements in an array : ");
    for(int i=0;i<5;i++)
    {
        scanf("%d", &arr[i]);
    }
    return arr;
}
```

DYNAMIC ARRAYS

- void* calloc(size_t num, size_t size)
  - reserves num * size bytes of memory
  - initializes memory to zero.

- void free(void *ptr) – gives the memory back
  - You are in charge of the memory used. Make sure to free anything you don't need.

DYNAMIC ARRAYS

```
// Allocate and return an array of 'n' integers
// Returns NULL on failure
// MUST RETURN A POINTER TO THE NEW MEMORY
int *myFunc(int n) {
        int i;

        // allocate block of memory
        int *x = malloc(n * sizeof(int));

        // test allocation for success
        if (x == NULL) {
                return NULL;
        }

        // fill it with zeros
        for (i=0; i<n; i++) {
                x[i] = 0;
        }

        // return the pointer
        return x;
}
```

USING DYNAMIC ARRAYS

```
int checkGetIntArrayAverageRandom()
{
        int n = rand() % 8 + 5;
        int *a = (int*)calloc(n,  sizeof(int));
        for (int i = 0; i < n; i++)
        {
                a[i] = rand() % 25 - 5;
        }
        double exp = getIntArrayAverageAns(a, n);
        int retVal = checkGetIntArrayAverage(a, n, exp);
        free(a);
        return retVal;
}
```

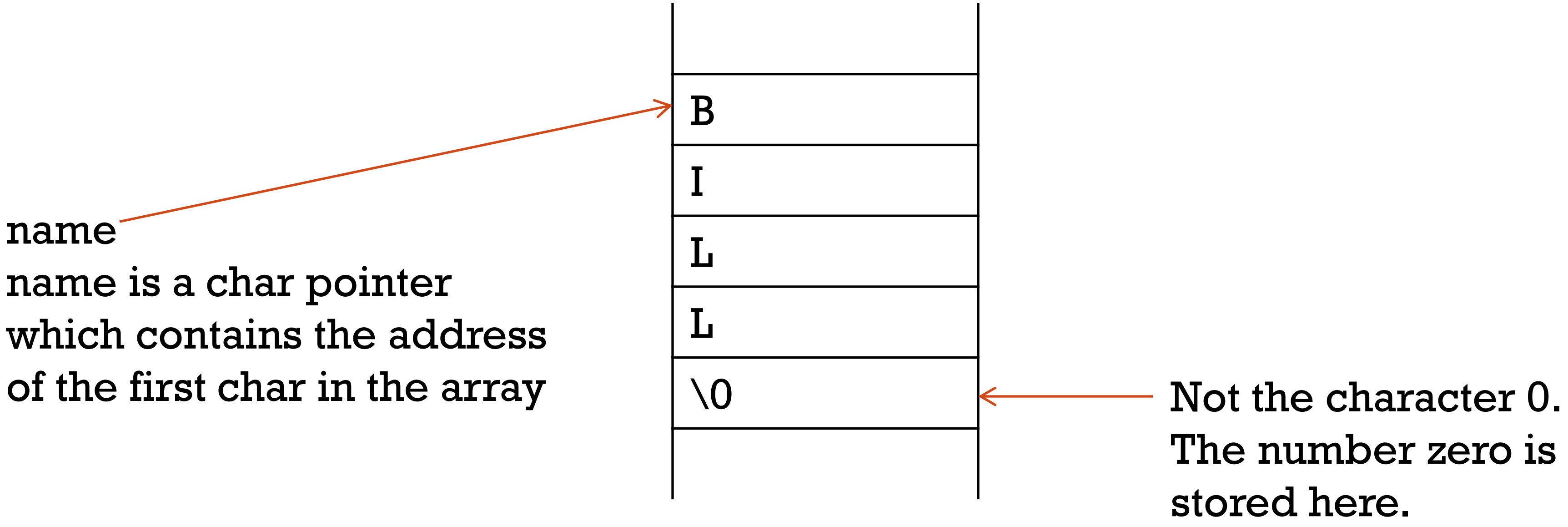
STRINGS

- Strings in C are arrays of characters.
- You must know the maximum length before creating a string so the array can be created of the correct size.
- Strings in C must be null terminated. That is the last character must be a null or zero (\0).
- You must reserve space for the null zero.

STRING EXAMPLE

char name[10] = "BILL"; //Null zero added

char *name = "BILL"; //Null zero added



SOME DIFFERENCES

```
char a[10] = “aaaaa”;
char  *b = “bbbbb”;
a = “ccccc”; //Not allowed
b = “ccccc”; //allowed
a++; //not allowed
b++; //allowed
a[5] = ‘x’; //allowed
b[5] = ‘x’; //not allowed
```

The last one will compile but will give segmentation fault in LINUX since string literals are stored as read only.

STRING POINTERS

- Strings literals are stored in their own special memory.

```
char *b =“123”;
char c = ‘X’;
b = “9999999999”;
printf(“%c\n”, c);
```

READING STRINGS WITH SCANF

```
char first[15];
char middle[15];
char last[20];

printf("Enter your first name: ");
scanf("%14s", first);  //The number tells scanf how many chars to read.

printf("Enter your middle name: ");
scanf("%14s", middle);

printf("Enter your last name: ");
scanf("%19s", last);

printf("%s, %s %s\n", last, first, middle);
```

SCANF OVERFLOW PROTECTION

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

char a[6] = "Hello";
char b = 'X';

int main()
{
    //Enter 12345, prints a=12345:b=X
    //Enter 123456 prints a=123456:b=        (X overwritten)
    //Enter 123456789 prints a=123456:b=     (6 chars max,  X overwritten)
    scanf("%6s", a);  //Should use a 5 here
    printf("a=%s:", a);
    printf("b=%c\n", b);
}
```

STRING LENGTH

- Part of <string.h> library
- size_t is simply an unsigned integer
- size_t strlen(const char *str)
  - Computes the length of the string str up to but not including the terminating null character.
- Can you write this function.

```
char a[15];
scanf("%14s", a);
int len = strlen(a); //Not including zero
```

STRING COPY

- char *strcpy(char *dest, const char *src)
  - Copies the string pointed to by src to dest.
- char *strncpy(char *dest, const char *src, size_t n)
  - Copies up to n characters from the string pointed to by src to dest.
  - When given a choice, always use the N version.

```
char a[10];
char b[10];
scanf("%9s", b);
strncpy(a, b, 9);
```

STRING CONCATENATION

- char *strcat(char *dest, const char *src)
  - Appends the string pointed to, by src to the end of the string pointed to by dest.
- char *strncat(char *dest, const char *src, size_t n)
  - Appends the string pointed to, by src to the end of the string pointed to, by dest up to n characters long.
  - When given a choice always use the N version

STRING COMPARE

- int strcmp(const char *str1, const char *str2)
  - Compares the string pointed to, by str1 to the string pointed to by str2.
- int strncmp(const char *str1, const char *str2, size_t n)
  - Compares at most the first n bytes of str1 and str2.
- These functions return values that are as follows:
  - if Return value < 0 then it indicates str1 is less than str2.
  - if Return value > 0 then it indicates str1 is greater than str2.
  - if Return value = 0 then it indicates str1 is equal to str2.

```
if (strcmp(s1, s2) == 0) {
        printf("Equal");
}
```

STRING SEARCH

- char *strstr(const char *haystack, const char *needle)
  - Finds the first occurrence of the entire string *needle* (not including the terminating null character) which appears in the string *haystack*.

- char *strrchr(const char *str, char c)
  - Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument str.

STRING TOKENIZING

- char *strtok(char *str, const char *delim)
  - Breaks string *str* into a series of tokens separated by *delim*.
  - Every time you call strtok, it returns a pointer to the next character after *delim*.
  - Returns null when no more tokens are available.

```
int main()
{
        char str[80] = "This is - www.tutorialspoint.com - website";
        const char s[2] = "-";
        char *token;

        /* get the first token */
        token = strtok(str, s);

        /* walk through other tokens */
        while( token != NULL )
        {
            printf( " %s\n", token );

            token = strtok(NULL, s);
        }

        return(0);
}
```

STRING FUNCTIONS EXAMPLE

```
char *s1 = "Tomorrow";
char s2[100] = "";
char *s3;

printf("Length of \"%s\" is %ld.\n",s1, strlen(s1));
strncpy(s2, "Testing", 100);
strncat(s2, " 1 2 3", 100 - strlen(s2));
printf("Length of \"%s\" is %ld.\n",s2, strlen(s2));


s3 = strchr(s1, 'r');
printf("s3 is %s.\n",s3);

s3 = strstr(s1, "ro");
printf("s3 is %s.\n",s3);
```

```
Length of "Tomorrow" is 8.
Length of "Testing 1 2 3" is 13.
s3 is rrow.
s3 is row.
Comparing "Tomorrow" and "Testing 1 2 3" gives 10
Comparing "Testing 1 2 3" and "Tomorrow" gives -10
Comparing "Tomorrow" and "Tomorrow" gives 0
```

```
printf("Comparing \"%s\" and \"%s\" gives %d\n",
      s1, s2, strncmp(s1, s2, 100));
printf("Comparing \"%s\" and \"%s\" gives %d\n",
      s2, s1, strncmp(s2, s1, 100));
printf("Comparing \"%s\" and \"%s\" gives %d\n",
      s1, "Tomorrow" , strncmp(s1, "Tomorrow", 100));
```

FUNCTION ORDER

- Functions should be declared before use in a file.
- In main, the call to printNum(1) causes a warning.
- If functions called are in another file (like our assignments) this will be an error.

```
void printNum(int n) {
        printf("%i\n", n);
}

int main() {
        printNum(1);
}
```

FUNCTION ORDER BETTER

- Functions should be declared before use in a file.
- Explicit declaration of a function using a prototype

```
void printNum(int n);  //Prototype
int main() {
		printNum(1);
}

void printNum(int n) {
		printf("%i\n", n);
}
```

PROTOTYPE EXAMPLE

- This is main.cc
- printChars and printRect are two functions declared in another file.
- But main has to know about them to compile correctly.

```
#include <stdio.h>
void printChars(int n, char c);
void printRect(int l, int w, char c);
int main()
{
        printChars(7,'X');
        printf("\n");
        printRect(5,8,'O');
        return 0;
}
```

FUNCTIONS AND PARAMETERS

- Pass by value
  - Argument is copied to new variable.
  - Changes in function do not affect original variable.
- Pass by reference
  - Argument is reference to the argument.
  - Changes refer to original variable and therefore changes are seen in the original.

BY VALUE VS BY REFERENCE

```
int main() {
        int x = 5;
        decArg1(x);
        printf("%d", x);
        decArg2(&x);
        printf("%d", x);
}
```

```
void decArg1(int x) {
        x = x - 1;
}
void decArg2(int *x) {
        *x = *x - 1;
}
```

STDLIB.H

- int atoi(const char *str)
  - converts string to integer.
- double atof(const char *str)
  - converts string to double
- int rand(void)
  - Returns a number from 0 to RAND_MAX
- void srand(unsigned in seed)
  - Seeds the random number generator
- void *malloc(size_in_bytes) and void *calloc(#items, size_of_one_item)
  - Assign or allocate dynamic memory
- free(void *ptr)
  - Deallocate memory
- void exit(int status)
  - Causes a normal program termination
- void abort(void)
  - Causes an abnormal program termination

RANDOM NUMBERS

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
    int main() {
        int i, n;
        time_t t;
        n = 5;
        srand((unsigned) time(&t));
        for( i = 0 ; i < n ; i++ ) {
            printf("%d\n", rand() % 50);
        }
        return(0);
    }
}
```