



CMSC 106

Lecture Set #2

Set Started:

Tuesday, September 4, 2007



C language – syntax

- *Syntax*

- *rules of the grammar*
- *vocabulary recognized by the language*
- *ANSI standard*
 - *American National Standards Institute*

- *Semantics*

- *the meaning of what is being said*



Syntax vs Semantics Examples

- The monster scared Jon.
 - syntactically valid
- Jon scared the monster.
 - syntactically valid
 - says something different than the first.
- Jon sat in the chair.
 - syntactically valid
- The chair sat in Jon.
 - syntactically valid
 - questionable in semantics
- The in sat. Chair Jon
 - Syntactically invalid
 - no semantic interpretation from this available at all



Program Errors

■ Incorrect Syntax

- The compiler gives error message at that spot and refuses to compile it.
- The compiler gives warning message at that spot but still compiles it.
- The compiler gives error or warning message at a spot later in the file.

■ Incorrect Semantics

- Program does nothing when run
- Program does nothing useful when run
- Program does the "wrong" thing when run
- Program "crashes" or "hangs" when running



Basic Program Structure

- a program must be comprised of 1 or more functions
 - function = named program part for performing a specific task
- must be 1 and only 1 function named **main**
 - controls everything else
 - starts there and determines who gets to go when
 - for now this is the only function you'll design



Functions: Definition and Use

- **Syntax of a function definition:**

```
func_type  func_name( list_of_parameters)
{
    func_body
}
```

Example of a complete program:

```
int main(){
    printf("this is a complete program\n");
}
```

Syntax of a function call

```
func_name(list_of_arguments);
```

Example of a function call:

```
printf("This is a complete program\n");
```



Functions: Identified by Name

■ Identifiers

- Used to name functions, variables, etc.
- String of alphabetic characters, numeric digits and the underscore
- Case sensitive
- Can not start with a numeric digit
- Must be a unique name



Identifier examples

- cmssc106
- CMSSC106
- cmssc_106
- cmssc.106
- _cmssc_106
- 106cmssc
- _106_cmssc_
- cmssc 106
- 106



printf function call details

```
#include <stdio.h>
int main(){
    printf("this is a complete program\n");
}
```

- printf writes its parameter/argument to the screen
- printf is defined in a library so it needs:
#include <stdio.h>
- The string argument to printf must be enclosed in " " (double quotes)
- prints string argument exactly as it appears - except escape sequences
 - i.e. \n (carriage return) which can appear anywhere between " "



Function Return

```
#include <stdio.h>
int main(){
    printf("this is a complete program\n");
    return 0;
}
```

- every function should end with a return statement that returns its "return value" to the caller
- main returns to the operating system
- 0 as a return value from main means "all is well"



Readability Issues

■ Comments

- `/* comment */`
- ignored by compiler
- for human reader
- multiple lines is fine
- used to explain what it is doing and/or how
- can not be nested
- every function needs a comment to tell its use and purpose
- every place it would help the readability comments should be included

■ Spacing

- vertical spacing
- horizontal spacing
- also ignored by the compiler and for the human reader
- should accurately reflect the meaning and flow



Spacing Issues

- white-space needed for readability (space, tab, end-of-line)
- Horizontal Spacing: INDENTING
- Vertical Spacing: BLANK LINES
- White-space does not matter to compiler
 - except between " "
 - except inside of words (or identifiers)
- Any amount of white space can appear between program components.
- Poor style
 - compiler doesn't care.
 - random indent statements.
 - lines longer than screen (or printer) width
 - if it doesn't accurately reflect program meaning or flow
- See examples on-line



Data types

- first two basic data types:

- **int**
- **float**

memory			
	+-----+		
		?	amount
	+-----+		
		?	grand_total
	+-----+		
		?	tax
	+-----+		
	+-----+		

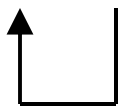
```
#include <stdio.h>
main()
{
    declarations
    statements
    return 0;
}
```

```
#include <stdio.h>
main(){
    int amount,
        grand_total;
    float tax;
    .
    .
    .
}
```

values random when first declared



Assignment Statement

- **variable name = value;**
- **Semantics**
 - when executed, the right side is calculated and
 - the result is stored in variable on left
- **a = b;**

- **previous value** of left-side variable is lost
- left side must be a variable
- right side can be:
 - a number (**literal**)
 - a **variable**
 - an **expression**
- right side's value unchanged (**copy**)

- e.g.:
grand_total = 125;

+	-----	+	
	?		amount
+	-----	+	
	125		grand_total
+	-----	+	
	?		tax
+	-----	+	
+	-----	+	



Initialization

- Combines Declaration and Assignment
- **Semantics**
 - When a variable space is first being allocated the value is immediately put in
 - a variable of the type and name given is created and given the value of the right hand side
- **int a = 6;**


A diagram showing a box containing the number 6. An arrow points from this box to the variable 'a' in the code 'int a = 6;'. Another arrow points from the variable 'a' to the text 'new variable comes into existence'.

 - **new variable comes into existence**
 - variable of that name can not already exist in the current scope
 - right side can be:
 - a number (**literal**)
 - a **variable**
 - an **expression**
 - right side's value unchanged (**copy**)

- e.g.:
int amount = 7,
grand_total = 125;
float tax = 0.05;

+-----+			
	7		amount
+-----+			
	125		grand_total
+-----+			
	0.05		tax
+-----+			
+-----+			



Example

- Trace this program's variables in memory

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

```
int main() {  
    int num = 3, num2;  
    num2 = 17;  
    num2 = num;  
    return 0;  
}
```

- =====
- **Problem Solving: Exchanging variables' values**

- any variable can only hold one value at a time
- assigning a value to a variable causes its previous value to be lost
- Must use a "temporary" variable to exchange.



Printf – The rest of the story

■ Printing Variables

- Syntax of printf:

printf("literal string"); or
printf("format control string", list of variables);

- format control string must have a **format specifier** for **each variable** in list
- format specifiers: **%letter**
- **%d** - print as an integer
- **%f** - print as a real number

■ **Example of printf for values:**

```
printf("num is %d\n and num2 is %d\n", num, num2);
```

(ex: printf.with.values.c)





More escape sequences

- **\n** new line
- **\t** advances cursor to next tab stop
- **\r** carriage return
- **\b** backspace
- **\a** beep
- **\"** **"**
- **** ****
- **%%** **%**

- examples:

- **printf("Jan\n\tPlane\n");**

- output: Jan

Plane

printf("\\""\n\");
"\n"



Symbolic constants

#define NAME value

- gives a name to a **constant** value
- `#define BOILING 212`
- **no semicolon because it is not a C statement is it handled by the preprocessor**
- convention: to distinguish constant names they are written in
- **all uppercase letters**
- **(ex: constants.c)**



Data Types

- Integer Family
 - char typically 8 bits
 - short typically 16 bits
 - int typically 32 or 64 bits
 - long typically 32 or 64 bits
 - long long typically 64 bits
 - All are signed by default, but can be made unsigned
 - unsigned int (typically, 0 to 4,294,967,295)
 - Literals
 - Decimal (255), Hex (0x255), signed (-255)
 - Character ('a', '\n')
 - Don't be stingy with size, when in doubt use a larger size
- Floating Point
 - float, double, long double
 - Literals (3.14159, 1E10, 25., 6.023e23)
- Character and String Literals
 - Character Literals: 'a', '9', '\n'
 - String Literals: "a long dull string", "\n", ""



sizeof and limits on types

- ANSI only specifies minimum amount of space for a specified type – not an exact
- sizeof()
 - operator – returns the number of bytes when passed a type or a variable as the operand (in parentheses
 - grace.umd.edu: it returns an unsigned int on one system and a long int on the other
- casting of types needed because of inconsistency
 - `printf("%d", (int)sizeof(float));`
 - cast does not modify the type of the operand – it just returns a value of the type indicated

(ex: testsizes.c)