C Basics

Introduction to the C Programming Language

Review

- Assembler Examples
  - AVR Registers
  - AVR IO
  - AVR Addressing Modes
  - Processor Review
  - State Machine examples

C History

- Began at Bell labs between 1969 and 1973
- Strong ties to the development of the UNIX operating system
  - C was developed to take advantage of byte-addressability where B could not
- First published in 1978
  - Called K&R C
    - Maximum early portability
    - A psuedo “standard” before C was standardized

The C Standard

- First standardized in 1989 by American National Standards Institute (ANSI)
  - Usually referred to C89 or ANSI C
- Slightly modified in 1990
  - Usually C89 and C90 refer to essentially the same language
- ANSI adopted the ISO.IEC 1999 standard in 2000
  - Referred to as C99
- C standards committee adopted C11 in 2011
  - Referred to as C11, and is the current standard
  - Many still developed for C99 for compatability

What is C?

- Language that “bridges” concepts from high-level programming languages and hardware
  - Assembly = low level
  - Python = Very high level
    - Abstracts hardware almost completely
- C maintains control over much of the processor
  - Can suggest which variables are stored in registers
  - Don’t have to consider every clock cycle
- C can be dangerous
  - Type system error checks only at compile-time
  - No garbage collector for memory management
    - Programmer must manage heap memory manually

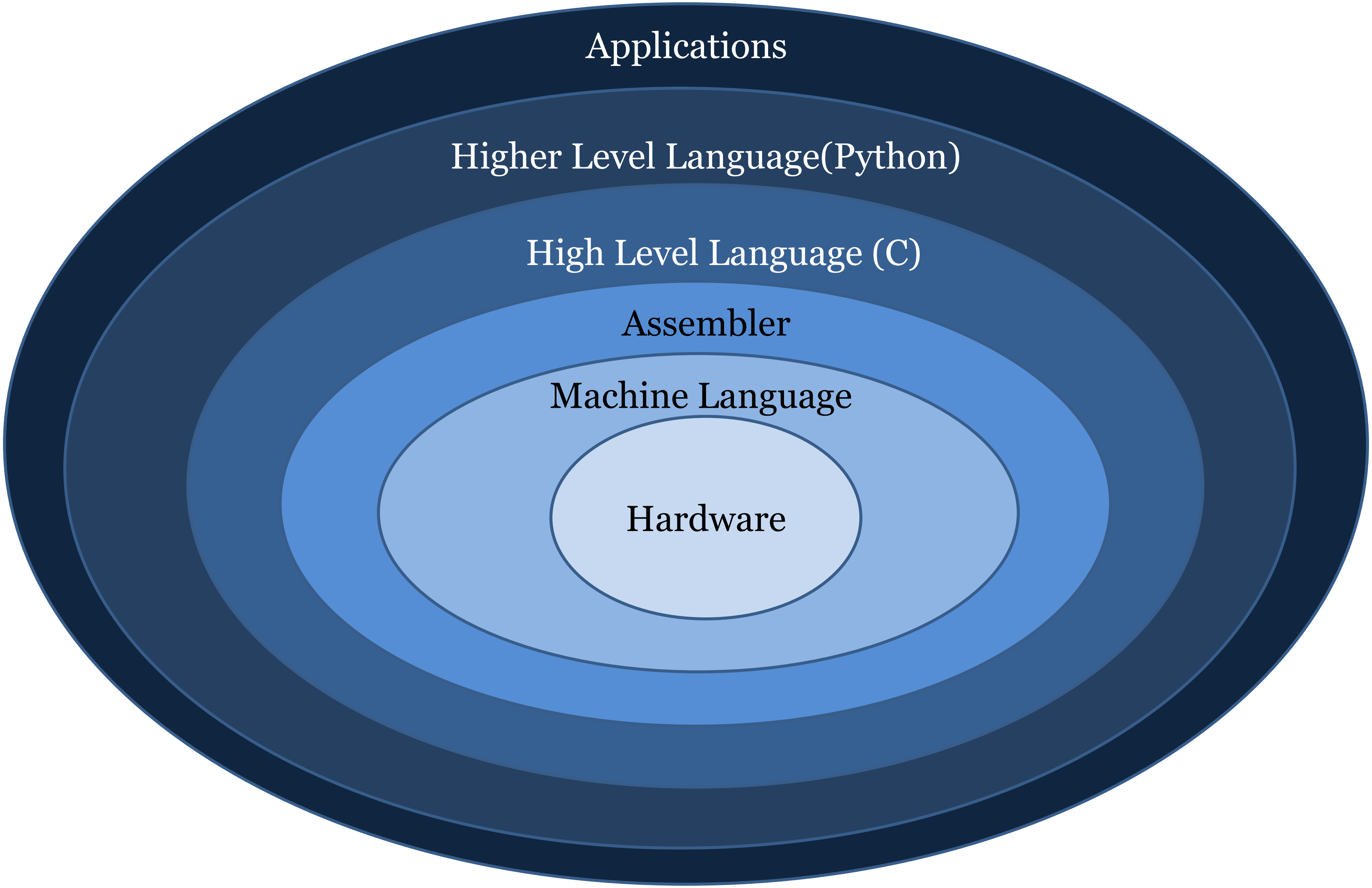
C Resources

- http://cslibrary.stanford.edu/101/EssentialC.pdf
- http://publications.gbdirect.co.uk/c_book/
- MIT Open Courseware
  - http://ocw.mit.edu/courses/#electrical-engineering-and-computer-science

C vs. Java

- C is a procedural language
  - Centers on defining functions that perform single service
    - e.g. getValidInt(), search(), inputPersonData()
  - Data is global or passed to functions as parameters
  - No classes
- Java and C++ are Object Oriented Programming languages
  - Centers on defining classes that model “things”
    - e.g. Sphere, Ball, Marble, Person, Student, etc…
    - Classes encapsulate data (instance variables) and code (methods)

Hardware to Application Onion Model



Libraries

- Library is composed of predefined functions
  - As opposed to classes for OOP language
  - Examples include:
    - Char/String operations (strcpy, strcmp)
    - Math functions (floor, ceil, sin)
    - Input/Output Functions (printf, scanf)
- C/Unix manual – “man” command
  - Description of C library functions and unix commands
    - e.g. “man printf” or “man dir”

Hello World

```
/*
file header block comment
*/
#include <stdio.h>
int main( )
{
        // print the greeting ( // allowed with C99 )
        printf( “Hello World\n”);
        return 0;
}
```

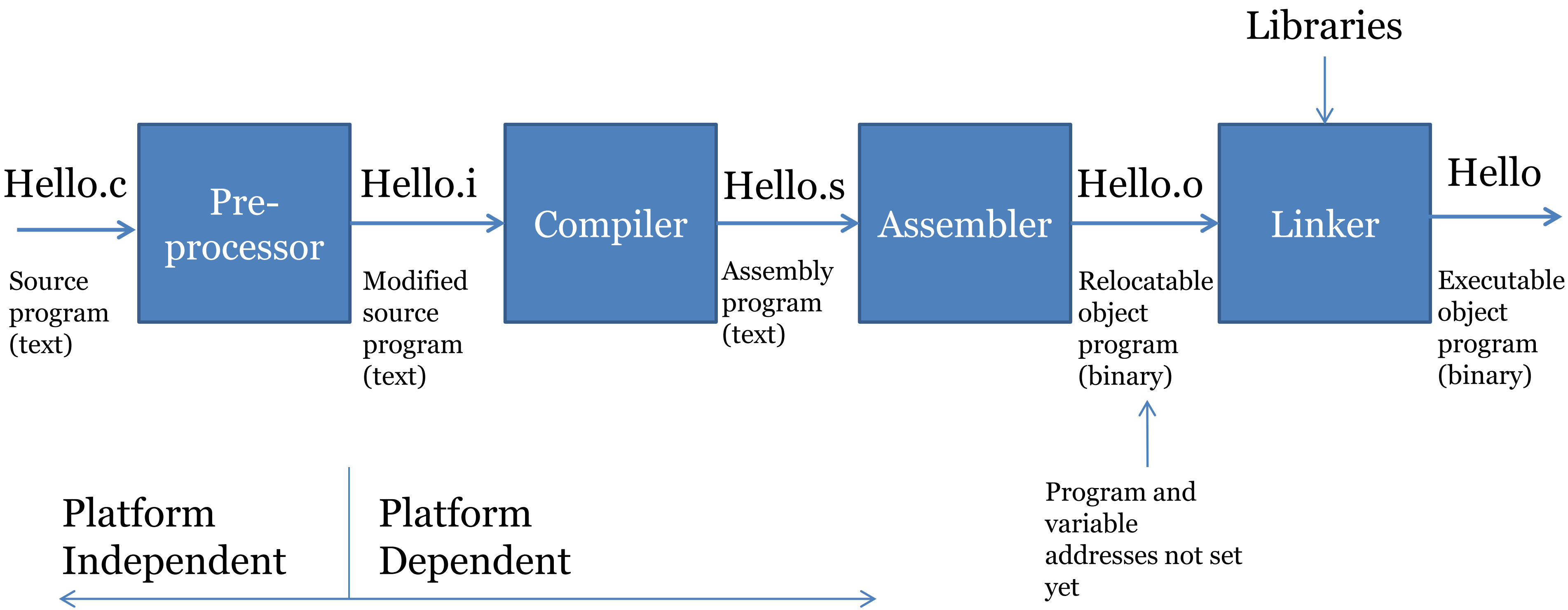
Compiling on Unix

- Traditionally the name of the C compiler that comes with Unix is “cc”
  - UMBC GL systems use the “GNU Compiler Collection”
    - “gcc” to compile C (and C++ programs)
  - Default name of executable program created by gcc is a.out
    - Can specify executable using -o command

Compiler Options

- -c
  - Compile only, don't link
    - Create a .o file, but no executable
  - E.g. gcc –c hello.c
- -o fname
  - Name the executable filename instead of a.out
  - E.g. gcc –o hello hello.c
- -Wall
  - Report all warnings
- -ansi
  - Enforce ANSI C standard, disable C99 features

Compilation Flow



Program is executed by calling name of executable at Unix prompt:
E.g. unix>hello

Compiler Vocabulary

- Preprocessor
  - Prepares file for compiler, handles processing macros, source selection, preprocessor directives, and file includes
- Compiler
  - Converts (nearly) machine independent C code to machine dependent assembly code
- Assembler
  - Converts assembly language to machine language of an object relocatable file (addresses not all resolved)
- Linker
  - Combines all object files and resolves addressing issues
- Loader
  - When executed, loads executable into memory
- Cross compiler
  - Compiler that runs on one platform but outputs code for another target machine (e.g. AVR is compiled on Intel)

Identifiers

- Identifier – name of a function or variable
- ANSI/ISO C standard
  - CASE SENSITIVE
  - First character must be alpha or _
  - May NOT be a C keyword such as int, return, etc…
  - No length limit imposed by standard
    - May have compiler limitation
- Good coding practices
  - Choose convention for capitalization of variables and functions
  - Symbolic constants should be all caps
  - Choose descriptive names over short names

Choosing Identifiers example

- T1, Temp1, Temperature1

- Which of the three above is most useful?

- Treat identifiers as documentation
  - Something which you would understand 3 years later
  - Don't be lazy with naming, put effort into documentation

Declaring, Defining, Initialization

- C allows you to declare and define variables
- A declaration puts the variables name in the namespace
  - No memory is allocated
  - Sets identifier (name) and type
- A definition allocates memory
  - Amount depends on variable type
- An initialization (optional) sets initial value to be stored in variable

C Declaration Example

- In C, combined declaration and definition is typical

```
char example1; //definition and declaration
int example2 = 5; //def. decl. and init.
void example3(void){   //def. and decl. of a function
    int x = 7;
}
```

- The “extern” keyword may be added to declare that definition will be provided elsewhere

```
extern char example1;
extern int example2;
void example3(void);
```

A function which does not provide definition is sufficient for the compiler.
This declaration is called a prototype

Assignments

- Assignments set values to variables
- Uses equal “=“ character and end with semicolon
  - E.g. temperature1 = 3;
  - temperature2 = temperature1;

Initialization

- Refers to the first assignment whether in declaration or afterward
- Until initialization, variables are considered uninitialized
  - Contents are unknown/unspecified/garbage
  - Exception: All objects with static storage duration (variables declared with static keyword and global variables) are zero initialized unless they have user-supplied initialization value
    - Still good practice to provide explicit initialization
- Initialization is not “free” in terms of run time or program space
  - Equivalent to a LDI

Types

- Intrinsic (fundamental, built-in) types
  - Integral types
    - E.g. int, char, long
  - Floating-Point types
    - E.g. float, double
- Type synonyms (aliases) using “Typedef”
  - Keyword “typedef” can be used to give new name to existing type
  - Example:

```
typedef unsigned int my_type;
my_type a=1;
```

  - Useful for Structures (covered later)

Integral Data Types

- C data types for storing integers are:
  - int (basic integer data type)
  - short int (typically abbreviated as short)
  - long int (typically abbreviated as long)
  - long long int (C99)
  - char (C does not have "byte")
  - int should be used unless there is a good reason to use one of the others
- Number of bytes
  - char is stored in 1 byte
  - Number of bytes used by other types depends on machine being used

Integral Type Sizes

- C standard is specifically vague regarding size
  - A short must not be larger than an int
  - An int must not be larger than a long int
  - A short int must be at least 16 bits
  - An int must be at least 16 bits
  - A long int must be at least 32 bits
  - A long long int must be at least 64 bits
- Check compiler documentation for specific lengths

Integral Specifiers

- Each of the integral types may be specified as:
  - Signed (positive, negative, or zero)
  - Unsigned (positive or zero only) (allows larger numbers)
- Signed is default qualifier
- Be sure to pay attention to signed vs. unsigned representations when transferring data between systems. Don't assume.

Common Embedded User Types

- To avoid ambiguity of variable sizes on embedded systems, named types that make size apparent should be used
- WinAVR has predefined custom types:
  - int8_t - signed char
  - uint8_t - unsigned char
  - int16_t - signed int
  - uint16_t - unsigned int
  - int32_t - signed long
  - uint32_t - unsigned long
- These are defined in inttypes.h using typedef command

Floating Point Types

- C data types for storing floating point values are
  - float – smallest floating point type
  - Double – larger type with larger range of values
  - long double – even larger type
- Double is typically used for all floating point values unless compelling need to use one of the others
- Floating point values may store integer values

Floating Point Type

- C standard is again unspecific on relative sizes
  - Requires float < double < long double

- Valid floating point declarations:

  tloat avg = 10.6;
  double median = 88.54;
  double homeCost = 10000;

Character Data Types

- C has just one data type for storing characters
  - Char – just 1 byte
  - Because only 1 byte, C only supports ASCII character set
- Example assignments:
  - char x = 'A';
    - Equivalent to : char x = 65;
    - ASCII character set recognizes 'A' as 65

Const qualifier

- Any of the variable types may be qualified as const
- const variables may not be modified by your code
  - Any attempt to do so will result in compiler error
  - Must be initialized when declared
    - E.g. const double PI = 3.14159;
    - const int myAge = 24;
    - Const float PI; //valid, PI=0
    - PI = 3.14159; //invalid

Sizeof()

- Because sizes of data types in C standard are vaguely specified, C provides sizeof() operator to determine size of any data type

- sizeof() should be used everywhere the size of a data type is required
  - Maintain portability between systems

Variable Declaration

- ANSI C requires all variables be declared at the beginning of the “block” in which they are defined
  - Before any executable line of code
- C99 allows variables to be declared anywhere in code
  - Like java and C++
- Regardless, variables must be declared before they can be used

Arithmetic Operators

- Arithmetic operators are the same as java
  - = : assignment
  - +,- : plus, minus
  - *,/,% : multiply, divide, modulus
  - ++, --: increment, decrement (pre and post)
- Combinations are the same
  - +=, -= : Plus equal, minus equal
  - *=, /=, %=: multiply equal, divide equal, mod equal

Boolean Data Type

- ANSI has no Boolean type
- C99 standard supports boolean data type
- To use bool, true, and false you must include stdbool.h

```
#include <stdbool.h>
Bool isRaining = false;
If(isRaining)
    printf(“Bring your umbrella\n”);
```

Logical Operators

- Logical Operators are closely similar in C and python and result in boolean value
  - &&: and
  - || : or
  - ==, !=: equal and not equal
  - <, <=: less than, less than or equal
  - >, >=: greater than, greater than or equal
- Integral types may also be treated as boolean expressions
  - 0 considered false
  - Any non-zero is considered true

Control Structures

- Both languages support the following control structures
  - For loop
  - While loop
  - Do-while loop
  - Switch statements
  - If and if-else statements
  - Braces ({,}) are used to begin and end blocks

Curly Braces

- Used to group multiple statements together
  - Control blocks, functions
  - Statements execute in order

```
int main(){
    int i=7;
    if(i==7) {
        i=i+j;
        int k;      //forbidden by c89 standard (c99 okay)
        k=i*I;      //variables declared at top of block
    }
}
```

If - Else block

```
if (expression) (statement)
    e.g. if(x>3) x+=1;  //simple form
if(expression) {  //simple form with {} to group
    statement;
    statement;
}
if(expression){    //full if/else form
    statement;
}  else  {
    statement;
}
```

If - Else If – Else block

```
if(expression1) {
   statement 1;
}  else if (expression2) {
   statement2;
}   else  {
   statement3;
}
```

Spacing Variation (Be Consistent)

```
if(expression) {
    statement;
}else {
    statement;
}
```

```
if (expression)
{
    statement;
}
else {
    statement;
}
```

```
if (expression)
{
    statement;
}
else
{
    statement;
}
```

There are many spacing styles for logic blocks. Pick one and **be consistent**.

Switch

```
switch (expression) {
   case const-expression-1:
      statement;
      break;
   case const-expression-2:
      statement;
      break;
   case <const-expression-3>: //combined case 3 and 4
   case <const-expression-4>:
      statement;
      break;
   case <const-expression-5>: //no break mistake? maybe
      statement;
   case <const-expression-6>:
      statement;
      break;
   default: // optional
      statement;
}
```

Omitting the break statements is a common error --it compiles, but leads to inadvertent fall-through behavior. This behavior is just like the assembly jump tables it implements.

While – Do While

```
while(expression){      //executes 0 or more times
    statement;
}

do{                     //executes 1 or more times
    statement;
}  while(expression)
```

For loops

```
for(initialization; continuation; action){
    statement;
}
for(; continuation; action){
    statement;
}
```

- Initialization, continuation and action are all optional.
- May optionally declare a variable in initialization (C99 standard)
- Continuation condition must be satisfied for every execution of statement, including the first iteration
- Action is code performed after the statement is executed

For loops

```
int i = 99;
for(; i!=0;){
    statement;
    i-=1;
}

for (int i = 99; i!=0; i=i-1){
    statement;
}
```

These are equivalent statements.

The second one is much more readable.

The second one also uses the C99 variable declaration inside the for loop. This may not work on AVR.

Break

```
while(expression){
    statement;
    statement;
    if(condition)
        break;
    statement;
    statement;
}
//control jumps here on break.
```

Continue

```
while(expression){
    statement;
    if(condition)
        continue;
    statement;
    statement;
    //control jumps here on continue
}
```

Conditional Expression

- Also called the Ternary Operator
- ?: (tri-nary "hook colon")
  - C: int larger=(x>y ? x:y);
  - Python: larger=x if x>y else y
- Syntax: expression1 ? expression2:expression3
  - Use this sparingly since it makes code less readable

Other Operators

- These operators are very similar in C and Java

- <<,>>,&,|,^ : bit operators
- <<=,>>=,&=,|=,^= : bit equal operators
- [] : brackets (for arrays)
- () : parenthesis for functions and type casting

- ^ - binary XOR

Arrays

- C supports arrays as basic data structure
- Indexing starts with 0
- ANSI C requires size of array be a constant
- Declaring and initializing arrays:

```
int grades[30];
int areas[10] = {1,2,3};
long widths[12] = {0};
int IQs[] = {120, 121, 99, 154};
```

Variable Size Arrays

- C99 allows size of array to be a variable

```
int numStudents = 30;
int grades[numStudents];
```

Multi-Dimensional Arrays

- C supports multi-dimensional array
- Subscripting provided for each dimension
- For 2-d arrays:
  - First dimension is number of "rows"
  - Second is number of "columns" in each row

```
int board[4][5];  // 4 rows, 5 columns
int x = board[0][0];  //1st row, 1st column
int y = board[3][4];//4th (last) row, 5th (last) column
```

#defines

- The #define directive can be used to give names to important constants
  - Makes your code more readable and changeable
- The compiler's preprocessor replaces all instances of the #define name with the text it represents
- Note, no terminating semi-colon

```
#define PI 3.14159
...
   double area = PI * radius * radius;
```

#define vs. const

- #define
  - Pro – no memory is used for the constant
  - Con – cannot be seen when code is compiled
    - Removed by pre-compiler
  - Con – not real variables, have no type
- const variables
  - Pro – real variables with a type
  - Pro – Can be examined by debugger
  - Con – take up memory

Examples

```
const int NUMBER = -42;
int main(){
    int x = -NUMBER;
}
```

If replaced with a # define, will throw compiler error

(-- is a decrement operator)

```
#define NUMBER 5+2
int x = 3 * NUMBER;
```

Value of x is 17 with #define, 21 with const

(int x = 3 * 5 + 2) vs (int x = 3 * 7)

```
#define NUMBER 5+2;
int x =  NUMBER * 3;
```

Compiler error

int x = 5 + 2; * 3;

Enumeration Constants

- C provides the *enum* as a list of named constant integer values (starting at 0 by default)
- Behaves like integers
- Names in enum must be distinct
- Often better alternative to #defines
- Example
  - Enum months{ JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
  - Enum months thisMonth;
  - thisMonth=SEP; //ok
  - thisMonth=42; //unfortunately, also ok

C Functions

- C Functions (no explicit procedures)
  - Have a name
  - Have a return type (a void return type represents a procedure)
  - May have parameters
- Before a function may be called, its “prototype” must be known to the compiler
  - Verify that function is being called correctly
  - Accomplished by:
    - Providing entire function prior to calling function in code
    - Provide function prototype prior to calling in code and providing function elsewhere

C Functions

- Unlike Java, a function is C is uniquely identified by its name
  - No concept of method overloading
  - There can only be one main() function in a C application
- UMBC coding standards dictate function names begin with UPPERCASE letter
  - E.g. AddThreeNumbers() instead of addThreeNumbers

Simple C Program

```
#include <stdio.h>
typedef double Radius;
#define PI 3.1415
/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius ){
   return ( PI * radius * radius );
}
// given the radius, calculates the circumference of a circle
double Circumference( Radius radius ){
   return (2 * PI * radius );
}
int main( ){
   Radius radius = 4.5;
   double area = circleArea( radius );
   double circumference = Circumference( radius );    // print the results
   return 0;
}
```

Simple C Program (prototypes)

```
#include <stdio.h>
typedef double Radius;
#define PI 3.1415
/* function prototypes */
double CircleArea( Radius radius );
double Circumference( Radius radius );
int main( ){
   Radius radius = 4.5;
   double area = circleArea( radius );
   double circumference = Circumference( radius );   // print the results
   return 0;
}
/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius ){
   return ( PI * radius * radius );
}
// given the radius, calcs the circumference of a circle
double Circumference( Radius radius ){
   return (2 * PI * radius );
}
```

Typical C Program

Includes

Defines, typedefs, data type definitions, global variable declarations, function prototypes

Main

Funciton Definitions

```c
#include <stdio.h>

typedef double Radius;
#define PI 3.1415

/* function prototypes */
double CircleArea( Radius radius );
double Circumference( Radius radius );

int main( )
{
   Radius radius = 4.5;
   double area = circleArea( radius );
   double circumference = Circumference( radius );

   // print the results
   return 0;
}

/* given the radius, calculates the area of a circle */
double CircleArea( Radius radius )
{
   return ( PI * radius * radius );
}

// given the radius, calcs the circumference of a circle
double Circumference( Radius radius )
{
   return (2 * PI * radius );
}
```