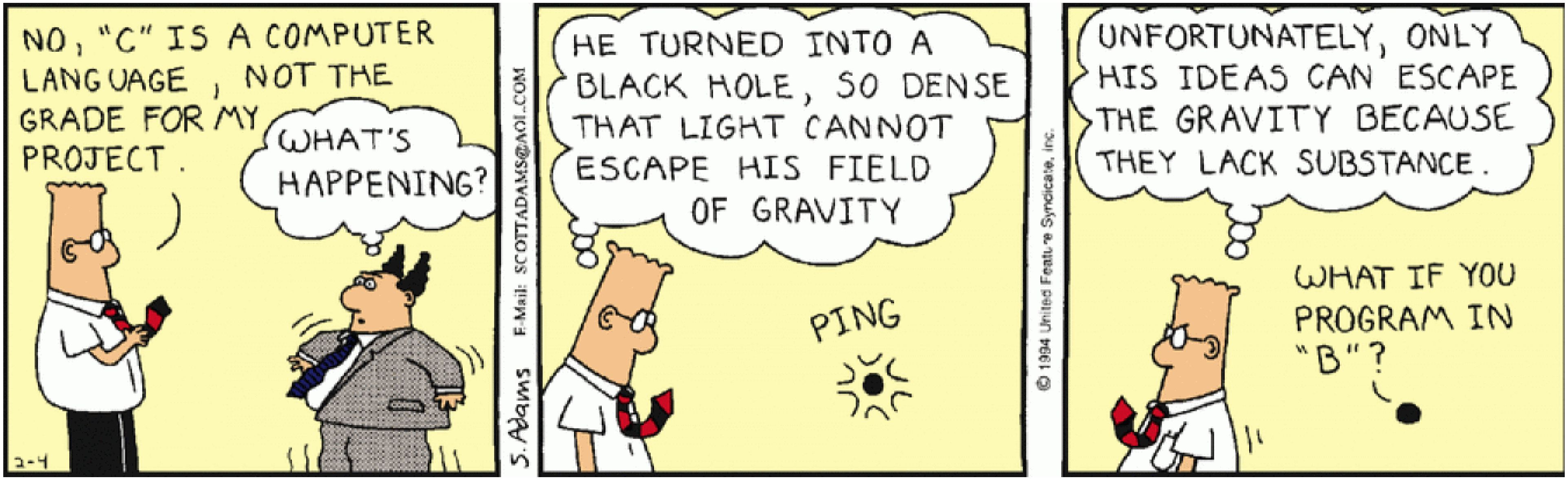
Applied Programming

Overview of the

C language


NO, "C" IS A COMPUTER LANGUAGE, NOT THE GRADE FOR MY PROJECT.
WHAT'S HAPPENING?
HE TURNED INTO A BLACK HOLE, SO DENSE THAT LIGHT CANNOT ESCAPE HIS FIELD OF GRAVITY
PING
UNFORTUNATELY, ONLY HIS IDEAS CAN ESCAPE THE GRAVITY BECAUSE THEY LACK SUBSTANCE.
WHAT IF YOU PROGRAM IN "B"?

2-4

S. Adams E-Mail: SCOTTADAMS@AOL.COM

Note: There was a "B" language before "C"

Applied Programming

- ANSI C89, C99 and C11
- Comments
- Program Organization (Modules)
- Standard Libraries
- Primitive Data Types

C Flavors

- Three major flavors
  - ANSI or C89 - the original
    - “High level” assembler
    - Must declare all variable first

http://americacomesalive.com/2015/07/20/ice-cream-cones-the-true-story/

  - C99
    - Adds some “modern” features
      - Floating declarations
      - “//” comments
  - C11
    - Adds a true “complex” data type

Semicolons “;”

- Each “C” statement ends with a **semicolon**.
  - Like most modern language (the all got it from C)
  - Forget one and the compiler will get angry ☺

I SAW THE CODE FOR YOUR COMPUTER PROGRAM YESTERDAY.
6-8
S. Adams
E-Mail: SCOTTADAMS@AOL.COM
IT LOOKED EASY. IT'S JUST A BUNCH OF TYPING. AND HALF OF THE WORDS WERE SPELLED WRONG.
© 1994 United Feature Syndicate, Inc.
AND DON'T GET ME STARTED ABOUT YOUR OVER-USE OF COLONS.
THEY REMIND ME OF YOU, SIR.

C89 (ANSI) vs C99

```
/* C89 must declare ALL variables first */
int i;
int j;
for(i = 0; i < 100; i++);
for(j = 0; j < 100; j++);
/* C++ “//” comments not allowed! */
```

-ansi or
-std=c89

```
// C99 can declare variables at any time
for( int i = 0; i < 100; i++);
for( int j = 0; j < 100; j++);
/* C++ style “//” comments  are allowed */
/* FYI: C++ is NOT C
  (they use a different compiler) */
```

-std=c99

Comments

- ANSI C89 must be bracketed between **`/*`** and **`*/`** and can span multiple lines

```
/* Comments in C can span multiple
   lines as shown here */
```

- C99 and C11 accept the C++ style comments

```
// C++ style comments are allowed
```

- *Use either the ANSI C89 or C99/C11standard*

Example: Comments

- What if we use **//** instead of **/*** and ***/** ?

```
/*
 * Applied Programming: ANCI C89 Comments
 *
 * Author: Juan C. Cockburn
 */

#include <stdio.h>

int main() {
   int a=12; // an integer

   printf("%d is an integer \n",a);
return(0); // is this necessary ?
}
```

- Compile using: (1) **`gcc`** , (2) **`gcc –Wall`**, (3) **`gcc –pedantic`**, (4) **`gcc -ansi`** . Note the different warning and error messages.

Example: Comments

Command	Result
gcc comments_ex.c	No errors or warnings
gcc -Wall comments_ex.c	No errors or warnings
gcc -pedantic comments_ex.c	comments_ex.c:10:13: warning: C++ style comments are not allowed in ISO C90 comments_ex.c:10:13: warning: (this will be reported only once per input file)
gcc -ansi comments_ex.c	comments_ex.c: In function 'main': comments_ex.c:10: error: expected expression before '/' token comments_ex.c:12: error: expected expression before '/' token

Primitives

- Like Java and Python

  - while …
  - do … until
  - if … else if … else …
  - switch
    case 1:
    case 2:
    default:

While

- Execute the enclosed loop until the “true” variable is non-zero.
    - Putting comments at the end of your blocks is ALWAYS a good idea
    - Follow some consistent block policy, neatness counts

```
while (true)
        {
        i++;
        } /* End while */
```

```
while (true) {
    i++;
 } /* End while */
```

Do while

- Execute the enclosed loop at least once, while the “true” variable is non-zero.
  - Both styles are good

```
do
  {
   i++;
  } while (true);
```

```
do  {
   i++;
} while (true);
```

If

- Execute the first block if “true” is non-zero, otherwise execute the else block
  - Always include the “{}” block delimiters

```
if ( true )
        {
        i++;
        }  /* end if */


else
        {
        i--;
        } /* end else */
```

```
if ( true ) {
   i++;
}  /* end if */

else {
    i--;
} /* end else */
```

Else If

- If statements can be chained

```
if ( true )
      {
      i++;
      }  /* end if */
else if (another)
      {
      i--;
      } /* end else if */
else
      {
      I = 5;
      } /* end else */
```

Curley brackets

- Most C commands don’t REQURE curly brackets “{“ and “}”

```
if ( true )
       i++;
i--
```

```
if ( true ) {
       i++;
    }
 i--;
```

Both are valid and work same

- Add to the “true” case: y++;

```
if ( true )
       i++;
       y++;
 i--;
```

```
if ( true ) {
       i++;
       y++;
    }
 i--;
```

NOT identical now!

- Curly brackets are **required** in this class

Switch

- Select different cases based on the value of the INTEGER variable.
  - Break statements required.

```
Switch (var) {
case 1:
   i++;
   break;
case 2:
   i--;
   break;
default :
    i = 5;
} /* End switch */
```

C89 Data Types

- Same as Java except
  - **boolean** in C - 0 is "false" anything else is true
  - **byte is char** in C
- In Java all types, except Boolean, are signed.

type	Java		C	
	bits	bytes	bits	bytes
boolean	1			
byte	8	1		
char	16	2	8	1
short int	16	2	16	2
int	32	4	16, 32, 64	2, 4, 6
long int	64	8	32, 64	4, 8
float	32	4	32	4
double	64	8	64	8
long double			80, 96, 128	10,12,16

Real Examples

VLSI machines (AMD Opteron(TM) Processor 6272, 1400 MHz)

```
Type     bits    bytes
-------------------------
  char          8        1
  short int    16        2
  int          32        4
  long int     64        8
  float        32        4
  double       64        8
long double 128         16
```

Raspberry pi (ARMv6-compatible processor rev 7, 795.44 MHz)

```
Type     bits    bytes
-------------------------
  char         8        1
  short int 16          2
  int         32        4
  long int    32        4
 float        32        4
 double       64        8
long double 64          8
```

Tip: In GNU/Linux use **`cat /proc/cpuinfo`** to get info about the CPU

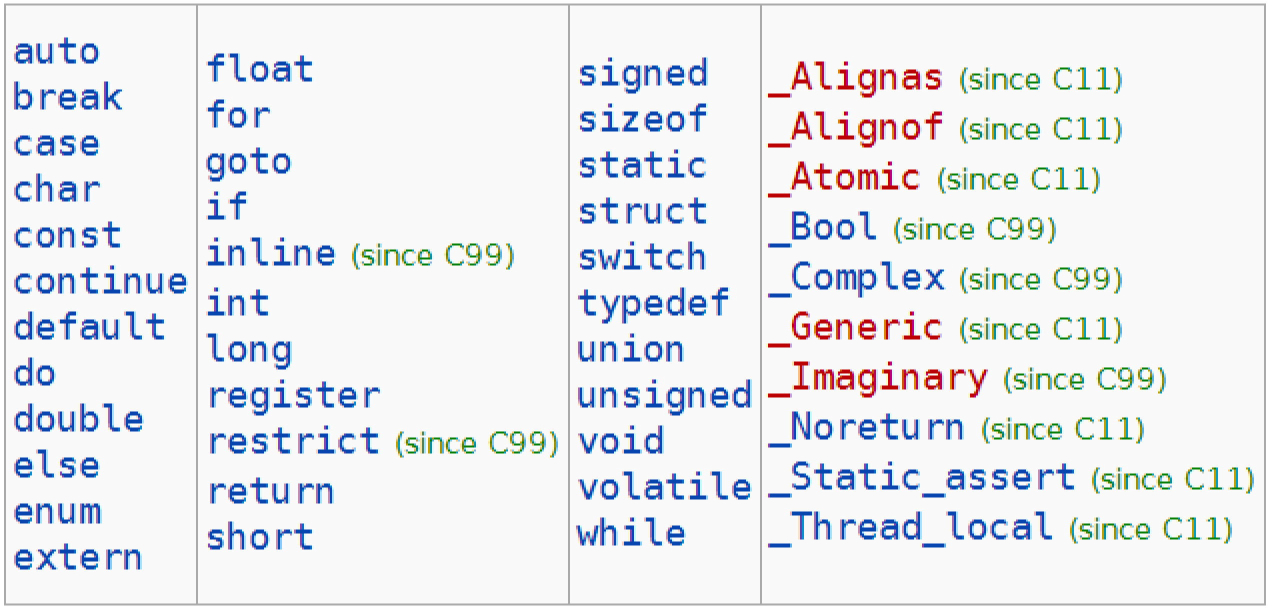
Problems with “C” types

- The classical C data type sizes can vary between machines, making porting code across platforms difficult.

- Most compilers support “stdint.h”

```
int8_t,  uint8_t
int16_t, uint16_t
int32_t, uint32_t
int64_t, uint64_t
```

ANSI C Keywords



auto
break
case
char
const
continue
default
do
double
else
enum
extern

float
for
goto
if
inline (since C99)
int
long
register
restrict (since C99)
return
short

signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while

_Alignas (since C11)
_Alignof (since C11)
_Atomic (since C11)
_Bool (since C99)
_Complex (since C99)
_Generic (since C11)
_Imaginary (since C99)
_Noreturn (since C11)
_Static_assert (since C11)
_Thread_local (since C11)

Source: en.cppreference.com/w/c/keyword

- C is a very compact language, there are only ***32 keywords in ANSI C89***

For a good online reference for the ANSI C89 library visit
http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

Special Keywords

Only limited use in this class (future)

- __volatile__
  - A command to the compiler that a variable can change at any time, used in interrupt driven code.
- __interrupt__
  - An instruction to the compiler that the function is an interrupt handler and to preserve all internal registers. No parameters can be passed in or out.
- static
  - When in front of a local variable, it allocates the variable in main memory so the variable persists through multiple function calls.

Primitive Data Types – C89

- Sample declarations and initialization

```
/* Integer Types primitive_declarations.c */
          char c = ’A’;    /* use single quotes */
          char n = -100;   /* char is a byte */
 unsigned char e = 100U;

             short =-2343;     /* no suffix */
              int  i=-2345678;
     unsigned int  ui = 100000000U;
              long li = 100000000L;
     unsigned long uli = 100000000UL;

/* Floating Point Types */
      float   pi = 3.1415926F;
     double    d = 0.123456789012345; /* no suffix */
long double ldpi = 0.31415e+1L;
```

Tip: The suffix (**`U,L,F`**) can also be lower case (it is only necessary for “constants”, *e.g.*, see **`math.h`** **`M_PIl`**)

eng-2500-06

```
Range of integer types
-----------------------------------------------
The minimum value of CHAR          = -128
The minimum value of SIGNED CHAR   = -128
The maximum value of CHAR          =  127
The maximum value of SIGNED CHAR   = +127
The maximum value of UNSIGNED CHAR =  255

The minimum value of SHORT INT          = -32768
The maximum value of SHORT INT          = +32767
The maximum value of UNSIGNED SHORT INT =  65535

The minimum value of INT          = -2147483648
The maximum value of INT          = +2147483647
The maximum value of UNSIGNED INT =  4294967295

The minimum value of LONG INT      = -9223372036854775808
The maximum value of LONG INT      = +9223372036854775807
The maximum value of UNSIGNED LONG = 18446744073709551615
```

C Program Organization

- C programs are organized as a ***collection of procedural functions***
  - ***variables*** must be ***declared before use***
  - C89 variables must be declared *at the beginning of a function block* (**`{ }`**) .
- ***Functions*** must be ***declared before use***
  - Function prototypes
- Execution starts with the function **`main()`**
  - similar to Java

Function Prototypes

- C only “knows” primitive data types and commands
  - Eg; int, for, while, etc.
- ALL other functions are NOT directly part of the language
  - printf, read, everything else!
  - Must “protoype” (define) a function before use

General Prototype form

<retType> FunName(<parm1>, (<parm2>, …);

RetType - The type of data the function will return
e.g. int, float, void, int *, etc

FunName - The name of the function
e.g. sin, log, etc

Parm1 - Parameters (variables) passed into the function. Zero to N parameters are allowed.
Normally a "fixed" list with a symbolic variable names
e.g. int angle, int *angle, float radian

; - Must end with a ";"

Examples

- A function that takes no parameters and returns no data

```
void Fun1(void);
```

- A function that takes two parameters and returns one. The first two are "better".

```
int  Fun2(int count, float *num);
int  Fun2(int count, float num);
int  Fun2(int, float);          /* not as good*/
```

return()

- Each C module can return a value or structure
- The “returned” type MUST match the data type defined in the function declaration.

- E.g:
    - int fun1() would use return(1);
    - float fun2() would use return(1.0);
    - char fun3() would use return(‘1’);

Note: return() “pops” up one level

exit()

- Exit is a special C “return” that terminates the current program and can pass a “return code” back to the operating system
- E.g: exit(99)
  - will terminate the current program at and set an operating system error code to 99
  - Useful for “crude” error handling

- Note: “return”ing from main() with a value is the same as “exit”ing from main.

The Standard Libraries

ANSI C89 has 15 Standard Libraries

http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

1. <assert.h> : Diagnostics
2. <ctype.h> : Character Class Tests
3. <errno.h> : Error Codes Reported by (Some) Library Functions
4. <float.h> : Implementation-defined Floating-Point Limits
5. <limits.h> : Implementation-defined Limits
6. <locale.h> : Locale-specific Information
7. <math.h> : Mathematical Functions*
8. <setjmp.h> : Non-local Jumps
9. <signal.h> : Signals
10. <stdarg.h> : Variable Argument Lists
11. <stddef.h> : Definitions of General Use
12. <stdio.h> : Input and Output*
13. <stdlib.h> : Utility functions*
14. <string.h> : String functions*
15. <time.h> : Time and Date functions

*important to us

ANSI C99 and C11

- C99 has 6 additional Standard Libraries

```
<complex.h> : for complex arithmetic
<fenv.h>    : for controlling IEEE-style floating
              point arithmetic
<inttypes.h>: for converting various types of integers
<stdbool.h> : for defining Boolean types and constants
<stdint.h>  : for defining integer types with size
              constraints
<tgmath.h>  : for defining type-generic math functions
```

- The most recent standard is C11 (2011.)
  - *In this course we will use C99*

C “modules”

- In large programming projects it is common to organize the code in ***files of related functions*** [similar to Python]
- A C module consists of two files:

(1) a header (“`.h`”) together with

– contains the module’s “public interface”

(2) a source code (“`.c`”) file.

– contains the “private implementation”

It is up to the programmer to organize their C program in a “modular” way

Example: Small C Program

```c
/* A Simple C Program 1 */
/* prog1ori.c */
int gcd(int a, int b) {
   if( 0 == b)
     return a;
   else
     return gcd(b, a % b);
}

int main() {
   int a,b ;
   a = 24; b = 40;
   printf("GCD: %d\n",
      gcd(a, b));
   return 0;
}
```

```c
/* A Simple C Program 2 */
/* prog2ori.c */
int gcd(int a, int b);

int main() {
   int a, b ;
   a = 24; b = 40;
   printf("GCD: %d\n",
      gcd(a, b));
   return 0;
}

int gcd(int a, int b) {
   if(0 == b)
     return a;
   else
     return gcd(b, a% b);
}
```

Which one runs ? Anything missing ?

Turning a C Program into a Module

```
/* A Simple C Program 2 */
int gcd(int a, int b)
```

This will become the include file (.h)

```
int main() {
   int a, b ;
   a = 24; b = 40;
   printf("GCD: %d\n",
      gcd(a, b));
   return 0;
}
```

This will become our “main”

Need to fix those missing braces too!

```
int gcd(int a, int b) {
   if(b == 0)
     return a;
   else
     return gcd(b, a% b);
   }
}
```

This will become our module C code

Cut it into pieces and add code (couple of lines) as needed

Small C Program as a Module

gcd_prog.c

```
/**
 ** Driver program to test
 ** gcd module
 **/

#include <stdio.h>
#include “gcd_module.h”

int main() {
   int a,b ;
   a = 24; b = 40;
   printf("GCD {%d, %d} : %d\n",
     a,b, gcd(a, b));
   return 0;
}
```

How do we compile it ?

gcc -ansi gcd_driver.c gcd_module.c -o gcd

gcd_module.h

```
/* “public” interface */
#ifndef _gcd_module_h_
#define _gcd_module_h_

int gcd(int a, int b);
#endif
```

gcd_module.c

```
/* “private” implementation      *
 * of recursive function to find  *
 * gcd of integers                */

#include “gcd_module.h”

int gcd(int a, int b) {
   if(b == 0)
     return a;
   else
     return gcd(b, a% b);
}
```

Important Detail

- In C we use #include to “include” the content of ***module headers***
  - including the standard libraries,
    #include <**stdio.h** >
    #include <**stdlib.h**>

- **`#include`** is **not**

```
     import numpy
     import java.io.*;
```

Why ?

Summary

- We will use mostly ANSI C99 (**`-ansi`**)
- C programs must be organized as sets of related functions or “modules” (**`foo.h`**, **`foo.c`**)
- You must declare all variables and all functions ***before they are used***.
- C is very “evil”, it assumes that the programmer knows what he/she is doing.
- A C program is not correct until it is free of all warnings (**`-Wall`**)

HW Hint 1 - Valgrind

- A FAMILY of tools to detect *memory management bugs, threading bugs* and profile programs in detail.
  - More in a future class

- Valgrind Sample:

```
valgrind --tool=memcheck --leak-check=yes ./bin [options]
```

  - **Where: ./bin** - binary to test
    options - any parameters or options for the binary

You always need to make sure that there are no memory leaks

Important: You must *compile with* the -g *option* (in gcc)

HW Hint 1

- Sample command:

```
valgrind --tool=memcheck --leak-check=yes ./qs   1 -1 1
```

- Sample output:

```
==27024== HEAP SUMMARY:
==27024==     in use at exit: 0 bytes in 0 blocks
==27024==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==27024== All heap blocks were freed -- no leaks are possible
```

- Goal: No leaks for any execution path in the code.
- Results can CHANGE for different execution paths

HW Hint 2 - gcc

- Typical command:

```
gcc –std=c99  –Wall –pedantic-O1 –g -lm  [files.c] –o binFile
```

-std=c99	- “using modern C”
-Wall	- all warnings
-pedantic	- more warning
-O1	- simple optimization, more warning
-g	- generate debug information
-lm	- include (-l) lib (m) math
files.c	- one or more .C files
-o binFile	- the name of the output binary is binFile

- Example:

```
gcc –stc=c99  -O1 –Wall –pedantic –g -lm QuadraticSolver.c –o qs
```

Exercise 1

- Describe what will happen in each case:

A:

```
int x = 1;
while (x) { x++;}
```

B:

```
int x = 0;
while (x) { x++;}
```

C:

```
int x = -1;
while (x) { x++;}
```

- Assuming a 32 bit integer:
  - A will loop about 2**32 times
  - B will not loop at all
  - C will loop one time

Exercise 2

- I, X & Y are all 1, VAR is zero.

```
if ( VAR )
        X++;
        Y++;
 I--;
```

What are the values of I, X & Y at the end?

- VAR is zero, or false so X++ never executes.
  Y++ is NOT affected by the if statement
  So: X = 1, Y = 2, I = 0

Exercise 3

- What, if anything, is wrong with this include (.H) file?

```
/* “public” interface */
#ifndef _gcd_module_h_
#define _gcd_module_h_

int gcd(int a, int b);
int x;
#endif
```

- int x; is CODE and does not belong in an include file

Appendix

Typical C program

```
#include <stdint.h>
uint32_t funct(uint32_t count);    /* prototype */
uint32_t globVar = 0x1;              /* global var */

/* All C programs start at main() */
int main(int argc, char* argv[] ) {
   uint32_t locVar;                  /* local variable */
   localVar = funct(globVar);
   While (localVar) { localVar--;}
return(0);
   } // End main
```

Primitive Data Types – C89

- Use **sizeof()** to find out how much memory (in bytes) each type takes (use the macro **CHAR_BIT** in **<limits.h>** to convert to bits:

```
/*****************************************
 * Memory space for intrinsic types in C *
 * primitive_types.c                     *
 * Author: Juan C. Cockburn               *
 *****************************************/
#include <stdio.h>
#include <limits.h>
int main() {
 printf("\n");   printf("A byte has %d bits\n\n", CHAR_BIT);
 printf("Memory space for intrinsic types in ANSI C \n\n");
 printf("  Type      bits     bytes\n");
 printf("---------------------------\n");
 printf(" char       %3lu      %3lu \n", sizeof(char)*CHAR_BIT,sizeof(char));
 printf(" short int  %3lu      %3lu \n", sizeof(short)*CHAR_BIT,sizeof(short));
 printf(" int       %3lu      %3lu \n", sizeof(int)*CHAR_BIT,sizeof(int));
 printf(" long int  %3lu      %3lu \n", sizeof(long)*CHAR_BIT,sizeof(long));

 printf(" float      %3lu     %3lu \n", sizeof(float)*CHAR_BIT,sizeof(float));
 printf(" double     %3lu     %3lu \n", sizeof(double)*CHAR_BIT,sizeof(double));
 printf(" long double %3lu    %3lu \n", sizeof(long double)*CHAR_BIT,sizeof(long double));

 return 0;
}
```

`short int` same as `short`
`long int` same as `long`

Range of Integer Types

```
#include <stdio.h>  /* for printf   integer_ranges.c */
#include <limits.h> /* info about ranges is here */
int main() {
   printf(“Range of integer types \n");
   printf("------------------------------\\nn");

   printf("The minimum value of SIGNED CHAR = %d\n", SCHAR_MIN);
   printf("The maximum value of SIGNED CHAR = %d\n", SCHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR = %d\n\n", UCHAR_MAX);

   printf("The minimum value of CHAR = %d\n", CHAR_MIN);
   printf("The maximum value of CHAR = %d\n", CHAR_MAX);
   printf("The maximum value of UNSIGNED CHAR  INT = %d\n\n", UCHAR_MAX);

   printf("The minimum value of SHORT INT = %d\n", SHRT_MIN);
   printf("The maximum value of SHORT INT = %d\n", SHRT_MAX);
   printf("The maximum value of UNSIGNED SHORT INT = %u\n\n", USHRT_MAX);

   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);
   printf("The maximum value of UNSIGNED INT = %u\n\n", UINT_MAX);

   printf("The minimum value of LONG = %ld\n", LONG_MIN);
   printf("The maximum value of LONG = %ld\n", LONG_MAX);
   printf("The maximum value of UNSIGNED LONG = %ld\n\n", ULONG_MAX);
return 0;
}
```

row_major.c

```
/*******************************************************
 * row_major.c
 * Example: Dynamically Allocated Matrices in row-major form
 *          allocated row by row and all at once
********************************************************/
#include <stdlib.h> /* for calloc and malloc */
#include <stdio.h>

/* Matrix element type goes here, change if necessary */
typedef int MatElement;

/* Functions to free memory space */
void free_rowmaj_matrix(MatElement **A,  int nr) {
  int i; /* row index */
  for (i=0; i<nr; i++)
    free(A[i]);
    A[i] = NULL;
  free(A);
  A = NULL;
}
void free_all_matrix(MatElement **A) {
  free(A[0]);
  A[0] = NULL;
  free(A);
  A = NULL’
}
```

```
/* Program begins */
int main() {

int k;
int i,j; /* row and column index counters */
int nr=6;   /* # of rows    */
int nc=3;   /* # of columns */

MatElement *ptr; /* Temporary pointer variable    */
MatElement **A;  /* Matrix, allocated row by row  */
MatElement **C;  /* Matrix, allocated all at once */

/* Allocate (row-major) matrix */
   A = malloc( nr * sizeof(MatElement *)); /*array of ptrs */
   for (i=0; i<nr; i++)
      A[i] = calloc( nc, sizeof(MatElement) );

/* Initialize matrix */
   k=0;
   for (i=0; i<nr; i++) {
      for (j=0; j<nc; j++){
        k+=1;
        A[i][j]=k;
      }
   }
```

```
 /* Allocate (row-major) matrix all at once */
   C = malloc( nr * sizeof(MatElement *));    /* array of ptrs */
   ptr = calloc( nr*nc, sizeof(MatElement) ); /* matrix elements */
   for (i=0; i<nr; i++) /* set row pointers properly */
      C[i] = ptr + nc*i;

/* Initialize matrix */
   k=0;
   for (i=0; i<nr; i++) {
      for (j=0; j<nc; j++){
        k+=1;
        C[i][j]=k;
      }
   }

/* Print results */
   printf("Comparing row by row allocation with all at once\n");
   printf("================================================\n");
   printf("Matrix elements take %lu bytes\n",sizeof(MatElement));
   printf("Elements stored in row-major form\n");
   printf("** Memory address in parenthesis\n");
   printf("\n");
/* Print Matrix */
   printf("Elements of A (row by row allocation)\n");
   for (i=0; i<nr; i++) {
      for (j=0; j<nc; j++)
        printf("%2d (%lu)",A[i][j],(unsigned long int)&A[i][j]);
      putchar('\n');
   }
```

```
printf("Total bytes: %d\n", (int)(&A[nr-1][nc-1]-&A[0][0])+1);
   putchar('\n');

/* Print Matrix */
   printf("Elements of C (all at once allocation) \n");
   for (i=0; i<nr; i++) {
      for (j=0; j<nc; j++)
        printf("%2d (%lu)",C[i][j],(unsigned long int)&C[i][j]);
      putchar('\n');
   }
   printf("Total bytes: %d\n", (int)(&C[nr-1][nr-1]-&C[0][0])+1);
   putchar('\n');

/* Clean up memory before end */

 free_rowmaj_matrix(A,nr);
 free_all_matrix(C);

 return 0;
}
```