C Essentials Training

3-day session

Overview	Understanding structural programming Understanding advance pointer and arrays Practical labs with GNU gcc compiler, gdb debugger
Duration	Three days - 24 hours (8 hours a day) 50% of lecture, 50% of practical labs.
Trainer	http://www.linkedin.com/in/pravinkumarsinha
Audience	Professional Software developers People supporting embedded and medium scale products.
Setup	Linux machine with GNU gcc compiler installed.

Lecture

Lecture session will be course content presentation through the trainer.
Any source code example related to the topic will be demonstrated, it would include executing the binaries.
Complete lecture material can be downloaded from
http://www.minhinc.com/training/advance-c-slides.pdf

Labs

Labs session would be completely hands on session where each example (with example data and execution instruction) would be provided to the students. Students can verify their results with the results provided in the material.

Day 1 Morning

Lecture - Introduction

- Source File
- Header File
- Object File
- Declaration and Definition
- Preprocessing, Compiling, Linking, Loading and Running
- Executable file format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Lecture - Data type, Operator and expression

- Data types and sizes
  -Integer
  -Floating point type
  -Pointer
  -Array
  -Enum
- Constant
- Variable Declaration and Definition
- Operator
  -Assignment operator and expression
  -Arithmetic operator
  -Relational operator
  -Conditional operator
- Bitwise operator
- Type conversion
- Conditional expression

Lecture - Array Pointers References

- Defining and initializing Array.
- Defining and initializing Pointers.
- Using pointers to access array elements.
- Pointers and const qualifiers.
- Dynamically allocated arrays.
- References.
- Independent references and restrictions.
- Multidimensional array argument to function

Day 1 Afternoon

Lab

- Write a c file and make static and shared library. Call library in main.
- Write a funtion to accept multidimensional array as an argument.
- Modify function argument to accept pointer to array.
- Implement a function to accept pointer to int and check were boundary overflow is not checked by the compiler.

Day 2 Morning

Lecture - Control structures

- Statement and Block
- Statements
  -If
  -Switch
  -While
  -For
  -do while
- Break and Continue
- Goto and Labels

Lecture - Function

- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference Parameters
- Recursion
- Inline function
- Preprocessor
  -File inclusion
  -Macro
  -Conditional inclusion

Lecture - Complex Data Types

- Struct
  -Structures and functions
  -Array of structure
  -Self referential structure
  -typedef
  -union
  -bit fields
- Array
- Pointers
- Ampersand operator(&)

Day 2 Afternoon

Lab

- Modify for loop to while loop
- Recursion
  - Implement fibonacci series
  - Implement factorial
- Check machine is little endian or big endian using union

Day 3 Morning

Lecture - Input and Output

- Standard Input and Output
- Buffered i/o
- Variable length argument list
- File access
- Line input and output
- Error handling - stderr and exit

Lecture - Storage class Specifier

- Automatic
- Const
- Global
- Extern
- Register
- Static
- Volatile

Lecture - Misc

- Debugging with gdb
- 64 bit data i/o

Day 3 Afternoon

Lab

- Try various buffering, character, line and file buffering. Check out speed differences.
- Modify global variable defined in one in other c file
  Build binary with -g option
  set break point on a function enter
  inside function print pointer to struct fields

C Essentials

C Essentials- Training Course

Minh, Inc.

Document modified on Sep-30-2019

Document contains 37 pages.

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_0

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_1

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_2

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_3

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

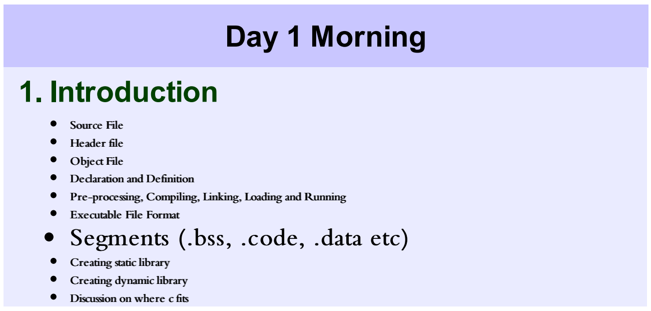
Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_4

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_5



Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

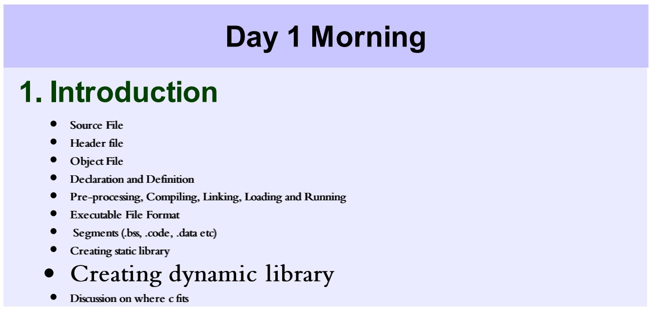
Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_6

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slidesgenda.php#chap1_7



Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

Refer:
http://www.minhinc.com/training/cpp/advance-cpp-slides.php#chap1_8

Day 1 Morning

1. Introduction

- Source File
- Header file
- Object File
- Declaration and Definition
- Pre-processing, Compiling, Linking, Loading and Running
- Executable File Format
- Segments (.bss, .code, .data etc)
- Creating static library
- Creating dynamic library
- Discussion on where c fits

C's type system and error checks exist only at compile-time. The compiled code runs in a stripped down run-time model with no safety checks for bad type casts,bad array indices, or bad pointers. There is no garbage collector to manage memory. Instead the programmer manages heap memory manually. All this makes C fast but fragile.

Perl and Java are more "portable" than C (you can run them on different computers without a recompile). Java and C++ are more structured than C. Structure is useful for large projects. C works best for small projects where performance is important and the programmers have the time and skill to make it work in C. In any case, C is a very popular and influential language. This is mainly because of C's clean (if minimal) style, it's lack of annoying or regrettable constructs, and the relative ease of writing a C compiler.

Day 1 Morning

2. Data type, Operator and expression

- Data types and sizes
  - Integer
  - Floating point type
  - Pointer
  - Array
  - Enum
- Constant
- Variable Declaration and Definition
- Operator
  - Assignment operator and expression
  - Arithmetic operator
  - Relational operator
  - Conditional operator
- Bitwise operator
- Type conversion
- Conditional expression

C provides a standard, minimal set of basic data types. Sometimes these are called "primitive" types.
More complex data structures can be built up from these basic types.

Integer Types
The "integral" types in C form a family of integer types.

- char ASCII character -- at least 8 bits. 8 bits provides a signed range of -128..127 or an unsigned range is 0..255. char is also required to be the "smallest addressable unit"

- short Small integer -- at least 16 bits which provides a signed range of -32768..32767. Typical size is 16 bits.

- int Default integer -- at least 16 bits, with 32 bits being typical. Defined to be the "most comfortable" size for the computer.

- long Large integer -- at least 32 bits. Typical size is 32 bits which gives a signed range of about -2 billion ..+2 billion. Some compilers support "long long" for 64 bit ints.

The integer types can be preceded by the qualifier unsigned which disallows representing negative numbers, but doubles the largest positive number representable.

Extra: Portability Problems
It is a good idea to use typedefs to set up types like Int32 for 32 bit int and Int16 for 16 bit int. It makes it processor architecture independent. Various typedef are available in stdint.h

<<stdint.h>

```
#ifndef __int8_t_defined
# define __int8_t_defined
typedef signed char             int8_t;
typedef short int               int16_t;
typedef int                     int32_t;
# if __WORDSIZE == 64
typedef long int                int64_t;
# else
__extension__
typedef long long int           int64_t;
# endif
#endif

/* Unsigned.  */
typedef unsigned char           uint8_t;
typedef unsigned short int      uint16_t;
#ifndef __uint32_t_defined
typedef unsigned int            uint32_t;
# define __uint32_t_defined
#endif
#if __WORDSIZE == 64
typedef unsigned long int       uint64_t;
#else
__extension__
typedef unsigned long long int  uint64_t;
#endif
.
.
.
```

char Constants

```
'A' uppercase 'A' character
'
' newline character
'	' tab character
'x00' the "null" character -- integer value 0 (different from the char digit '0')
'\012' the character with value 12 in octal, which is decimal 10
```

Integer constants
Numbers in the source code such as 234 default to type int. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a long such as 42L. An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal -- 0x10 is way of expressing the number 16. Similarly, a constant may be written in octal by preceding it with "0" - 012 is a way of expressing the number 10.

Type Combination and Promotion
The integral types may be mixed together in arithmetic expressions since they are all basically just integers with variation in their width. For example, char and int can be combined in arithmetic expressions such as ('b' + 5), int and long like (5+10L).

Pitfall -- int Overflow
(k*1024) may not work when k is int in low address bit memory. Way to fix the code is to   rewrite it as (k * 1024L) -- the long constant forced the promotion of the int.

Floating point Types
float Single precision floating point number typical size: 32 bits double precision floating point number typical size: 64 bits long double possibly even bigger floating point number (somewhat obscure)

Constants in the source code such as 3.14 default to type double unless the are suffixed with an 'f' (float) or 'l' (long double). Single precision equates to about 6 digits of precision and double is about 15 digits of precision.

floating point numbers is that they are inexact (1.0/3.0 + 1.0/3.0 + 1.0/3.0) // is this equal to 1.0 exactly?
- Do not Use equality (==) -- use inequality (<) comparisons instead.

Array
An array is a data structure which can store a fixed-size sequential collection of elements of the same type.

```
score
 |
 v
0       1       2       3                  99     100
--------------------------------           -----------
|      |       |       |          . . . .  |      |   x <-do not use
--------------------------------           -----------
int score[100];

Array declaration

- type arrayname[arraysize]
float a[5];
int x,y,b[5];

Array Initialization
int a[5]=(1,3,5,};//rest initialized with 0
int a[]={1,3,5,7,9};
int age[2][3]={{4,8,12},{19,6,-1}}


           0th col     1st col        2nd col
         --------------------------------------
0th row  |           |            |
         --------------------------------------
1st row  |           |            |
         --------------------------------------

0th row                                     1st row
0th col     1st col     2nd col             0th col        1st col       2nd col
--------------------------------------      --------------------------------------
|          |           |            |       |           |           |           |
--------------------------------------      --------------------------------------
```

Passing array argument
```
int func(int score[2][3])
int func(int score[][3]){}
int func(int (*score)[3]){}
func(a);//pass by address, value changed in func
```

Pointer

A pointer is a value which represents a reference to another value sometimes known as the pointer's "pointee".

Pointer operators

```
 * (Deference operator)   - means "the value of"
 & (address-of operator)   - means "address of"

 int a=5;

        5000|               |
    a-> 5004|     5         |<----+
                                  |
 int *pa=&a;                |     |
        6000|               |     |
   pa-> 6004|   5004        |<----+

struct fraction {
int numerator;
int denominator;
};


struct fraction *f1, *f2;



                                               -------------
      -------------                            |     7     | denominator
 f1 |             |  -------------+            +-----------+
      -------------               |            |     22    | numerator
               ^                  +----------->  -------------
               |                                  /
    struct fraction*                            struct fraction
                                               (the whole block
                                               of memory)
```

expression	Type
`f1`	`struct fraction*`
`*f1`	`struct fraction`
`(*f1).numerator`	`int`

complex declarations

```
struct fraction** fp; a pointer to a pointer to a struct fraction
struct fraction fract_array[20]; array of 20 struct fraction
struct fraction *fract_ptr_array[20]; an array of 20 pointers to struct fraction
struct fraction (*pfa)[100];
struct fraction (*(*pf)())();

f(int daytab[2][13]) { ... }

It could also be
f(int daytab[][13]) { ... }

Since the number of rows is irrelevant, or it could be
f(int (*daytab)[13]) { ... }

Which says that the parameter is a pointer to an array of 13 integers.
The parentheses are necessary since brackets [] have higher precedence
than *. Without parentheses, the declaration would be

int *daytab[13] // array of pointer to ints
```

There is an important difference between these definitions:

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
char (*(*x())[])()
x: function returning pointer to array[] of pointer to function returning char
char (*(*x[3])())[5]
x: array[3] of pointer to function returning pointer to array[5] of char
```

Dynamic allocation

```
malloc : memory allocation
free      : memory deallocation

int *intptr;
char *charpt;
intptr=(int*)malloc(sizeof(int));
charptr=(char*)malloc(sizeof(char)*6);
free(intptr);
free(charptr);
```

Memory leak - Memory allocated is not de-allocated.
Dangling pointer - A pointer that points to a variable that has been de-allocated.

Pointer arithmetic

```
int *pi;
char *pc;
++pi;//jumps 4 bytes in 32 bit processor
++pc;//jumps 1 byte
```

Array & Pointers

```
int a[10];
int *b=&a[0];

const char *pc="string";
char ac[]="string";

           ------------------------------------
pc ------->|
           ------------------------------------
```

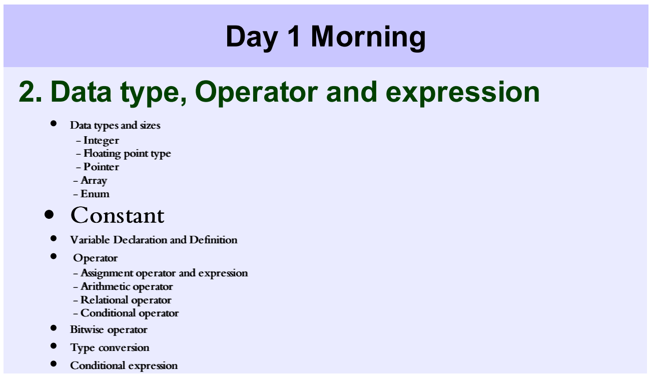
Pointer to a block of memeory

```
ac
----------------------------------
|
-----------------------------------
Array of memory fragments. No address
```

Enum

An enumeration is a list of constant integer values.

```
enum boolean { NO, YES };
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
JUL, AUG, SEP, OCT, NOV, DEC };
/* FEB = 2, MAR = 3, etc. */
eum {No,YES}; // global

boolean value=NO;

$cat main.c
#include <stdio.h>
struct SomeItem{
enum {MOVIE,MUSIC} itemType;
union{
int movieid;
char moviename[255];
};
};


int main(int argc, char *argv[]){
struct SomeItem someitem;
someitem.itemType=MUSIC;
someitem.movieid=10;
return 0;
}
```



Day 1 Morning

2. Data type, Operator and expression

- Data types and sizes
  - Integer
  - Floating point type
  - Pointer
  - Array
  - Enum
- Constant
- Variable Declaration and Definition
- Operator
  - Assignment operator and expression
  - Arithmetic operator
  - Relational operator
  - Conditional operator
- Bitwise operator
- Type conversion
- Conditional expression

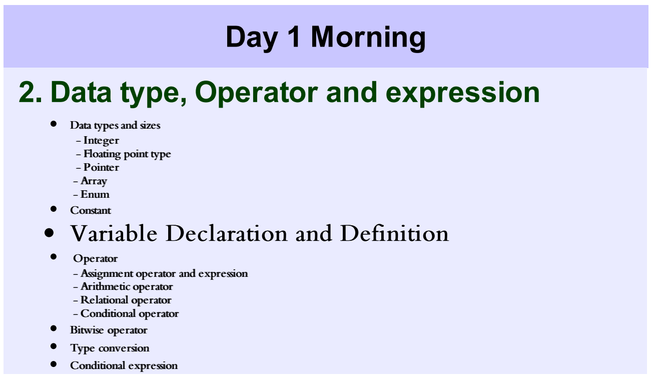
Const

The qualifier const can be added to the left of a variable or parameter type to declare that the code using the variable will not change the variable.

```
void foo(const struct fraction* fract);
int i, *pi, *const cpi = &i;
const int j=func();
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
const int *p; // pointer to constant
int * const p;//const pointer to integer
const int * const p;//const pointer to integer const
```

Const must be defined or initialized at the time of declaration.

Const has internal linkage similar to static.



Day 1 Morning

2. Data type, Operator and expression

- Data types and sizes
  - Integer
  - Floating point type
  - Pointer
  - Array
  - Enum
- Constant
- Variable Declaration and Definition
- Operator
  - Assignment operator and expression
  - Arithmetic operator
  - Relational operator
  - Conditional operator
- Bitwise operator
- Type conversion
- Conditional expression

Declaration - Asserts the existence of a variable, function or type defined elsewhere in the program. A variable may be declared by preceding its type with the keyword extern.

const declaration are file scope and must be initialized or defined at the time of declaration.

static variable and functions are file scoped and they can not be defined in other files like extern.

$ cat a.c
```
extern int a;
extern const int b=10;
```

$ cat b.c
```
#include <stdio.h>
int a=10;//defined here
const  int b=20; // error while linking
int main(){
return 0;
}
```

$ gcc -c a.c -o a.o
```
a.c:2: warning: 'b' initialized and declared 'extern'
```
$ gcc -c b.c -o b.o

$ gcc -o aa a.o b.o
```
b.o:(.rodata+0x0): multiple definition of 'b'
a.o:(.rodata+0x0): first defined here
collect2: ld returned 1 exit status
```

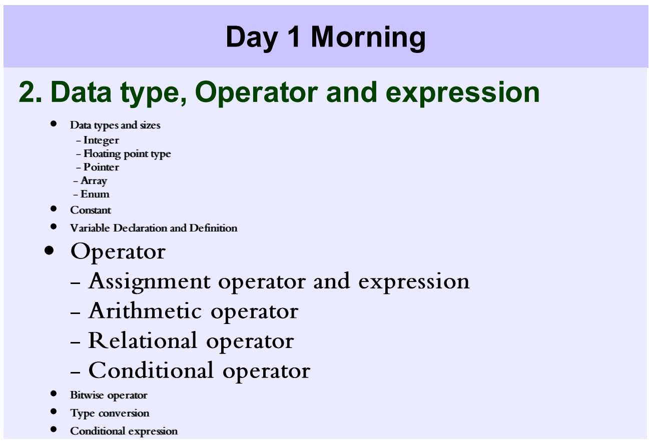
```
int a; // just declaration
void func();//just declaration

c++
int a // declaration and definition
extern a // declaration
```

Declaration can be made multiple times.

Definition - Allocates storage for a variable of a specified type and optionally initializes the variable

```
int a=10; //declaration and definition
void func(){// declaration and definition
}
```

- Definition must be only once.



Day 1 Morning

2. Data type, Operator and expression

- Data types and sizes
  - Integer
  - Floating point type
  - Pointer
  - Array
  - Enum
- Constant
- Variable Declaration and Definition
- Operator
  - Assignment operator and expression
  - Arithmetic operator
  - Relational operator
  - Conditional operator
- Bitwise operator
- Type conversion
- Conditional expression

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

- Assignment operator and expression
Here's the list of assignment shorthand operators...

```
+=, -= Increment or decrement by RHS
*=, /= Multiply or divide by RHS
%= Mod by RHS
>>= Bitwise right shift by RHS (divide by power of 2)
<<= Bitwise left shift RHS (multiply by power of 2)
&=, |=, ^= Bitwise and, or, xor by RHS

- Arithmetic Operators
+ Addition
- Subtraction
/ Division
* Multiplication
% Remainder (mod)

- Unary Increment Operators: ++ --
```

The unary ++ and -- operators increment or decrement the value in a variable.

```
var++ //post variant
++var //pre variant
```

- Relational Operators
These operate on integer or floating point values and return a 0 or 1 boolean value.

```
== Equal
!= Not Equal
> Greater Than
< Less Than
>= Greater or Equal
<= Less or Equal
```

Problem.

```
while(x=3);//legal
while(x==3);//legal
```

- Logical Operators
The value 0 is false, anything else is true.

```
! Boolean not (unary)
&& Boolean and
|| Boolean or
```

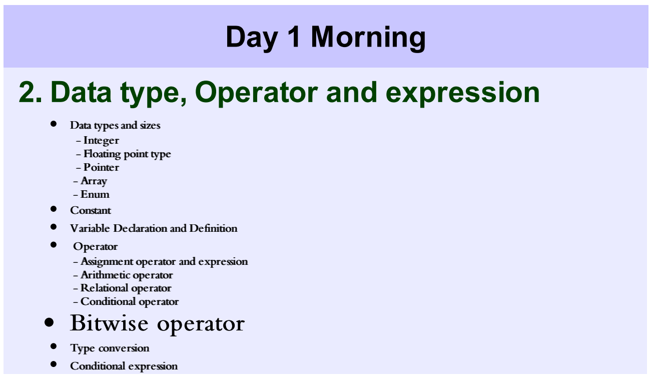
- Conditional Operators ?:

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions.

- Misc Operators

```
sizeof() Returns the size of an variable
& Returns the address of an variable
* Pointer to a variable
```



Day 1 Morning

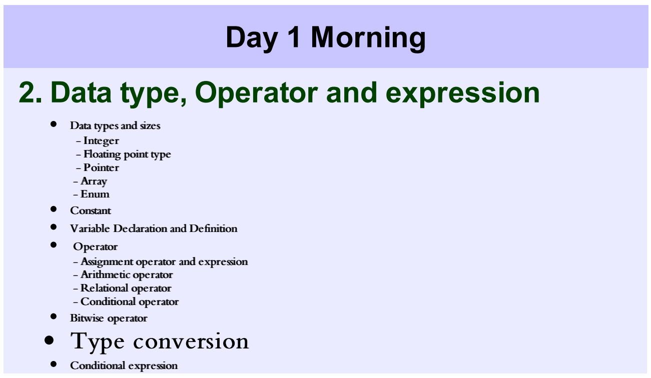
2. Data type, Operator and expression

- Data types and sizes
  - Integer
  - Floating point type
  - Pointer
  - Array
  - Enum
- Constant
- Variable Declaration and Definition
- Operator
  - Assignment operator and expression
  - Arithmetic operator
  - Relational operator
  - Conditional operator
- **Bitwise operator**
- Type conversion
- Conditional expression

Bitwise Operators
C includes operators to manipulate memory at the bit level.

```
~ Bitwise Negation (unary)  flip 0 to 1 and 1 to 0 throughout
& Bitwise And
| Bitwise Or
^ Bitwise Exclusive Or
>> Right Shift by right hand side (RHS) (divide by power of 2)
<< Left Shift by RHS (multiply by power of 2)
```

- Note & and && are different
```
unsigned int a = 60; /* 60 = 0011 1100 */
unsigned int b = 13; /* 13 = 0000 1101 */
int c = 0;
c = a & b; /* 12 = 0000 1100 */
printf("Line 1 - Value of c is %d
", c );
c = a | b; /* 61 = 0011 1101 */
printf("Line 2 - Value of c is %d
", c );
c = a ^ b; /* 49 = 0011 0001 */
printf("Line 3 - Value of c is %d
", c );
c = ~a; /*-61 = 1100 0011 */
printf("Line 4 - Value of c is %d
", c );
c = a << 2; /* 240 = 1111 0000 */
printf("Line 5 - Value of c is %d
", c );
c = a >> 2; /* 15 = 0000 1111 */
```



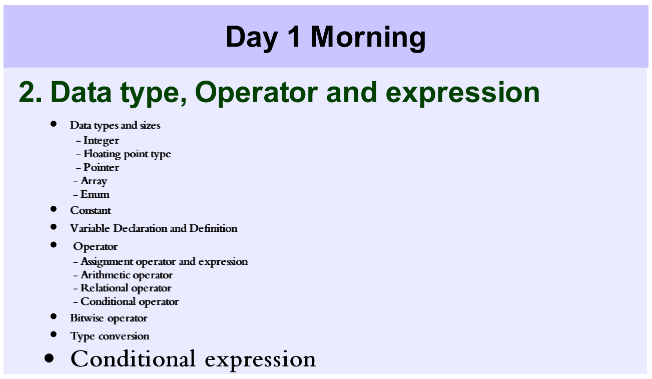
Day 1 Morning

2. Data type, Operator and expression

- Data types and sizes
  - Integer
  - Floating point type
  - Pointer
  - Array
  - Enum
- Constant
- Variable Declaration and Definition
- Operator
  - Assignment operator and expression
  - Arithmetic operator
  - Relational operator
  - Conditional operator
- Bitwise operator
- **Type conversion**
- Conditional expression

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a promotion "narrower" operand into "wider" one without losing information, such as converting an integer into floating point in an expression like f + i.

Truncation
Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.



Day 1 Morning

2. Data type, Operator and expression

- Data types and sizes
  - Integer
  - Floating point type
  - Pointer
  - Array
  - Enum
- Constant
- Variable Declaration and Definition
- Operator
  - Assignment operator and expression
  - Arithmetic operator
  - Relational operator
  - Conditional operator
- Bitwise operator
- Type conversion
- Conditional expression

Conditional Expression ?:
Exp1 ? Exp2 : Exp3;
Where Exp1, Exp2, and Exp3 are expressions.

3. Array Pointers References

- Defnining and Initialzing Array
  - Defining and initializing Pointers
  - Using Pointers to access array elements
  - Pointers and const qualifiers
  - Dynamic allocated arrays
  - Multidimensional array arugment to function

- Define and initialize static array
- Array has to be initialized with constant variable that is qualified at compile time

```
int ia[get_size()]  // get_size() is function and resolved at run time
const int size=get_size(); // size resolved at run time
int ia[size]; // error
int ia[10*2-10]; // resolved at compile time
```

- Local copy are uninitialized where as global copy is initialized to 0.
- Uninitialized stati c or global array goes to .bss section in the binary.
- Array is initialized in {} block

```
int arr[]={1,2,3} // array size 3
```
or
```
Int arr[3]={1,2,3}
Int arr[5]={1,2,3} // rest initialized with 0
```

Uninitialized const array takes garbage value
```
const int arr[3];// garbage value

C++ const uninitialized array is an error
const int arr[3]={1}; // rest initialized with 0
const int *arr=new arr[3]() // c++ value initialized
```

Breaking array boundary is not an exception and it memory over writing
```
int ary[1];
ary[2]=0;
```

Function array arguments are actually pointer to array.
```
void func(int ap[4]);
Void func(int ap[]);
void func(int *ap); //all are same
func (arr);
```

-Pointer to array and multidimensional array
```
void func(int (*arr)[10]);
void func(int arr[][10]);
void func(int arr[4][10]);// all same
```

- Array can not be copied or assigned
```
int ia2[](ia); // error
ia2 = ia1; //error
```

- Array size can not be known.
```
sizeof(arr)/sizeof(&arr[0])
```

- Array memory management is in users hand. Chance of memory overwriting.

```
void func(int pa[6]){
pa[6]=8;
}
int pa[4];
func(pa);
```

- Character array can be initialized with string null terminated.
```
char ca[] = "C++";
```

- Character array can be initialized character wise, many not be null terminated
```
char cal[] = {'C', '+', '+', '\0' };
```

- In case of character strings use strn string function for manipulation, si.e
```
strncpy, strncat.
```

Day 1 Morning

3. Array Pointers References

- Definining and Initialzing Array
- **Defining and initializing Pointers**
- Using Pointers to access array elements
- Pointers and const qualifiers
- Dynamic allocated arrays
- Multidimensional array arugment to function

```
type * pointer_name[,*pointer,name2,..];
char * terry = "hello"; //character string
const char *terry;//pointer to constant char
char *const terry;//const pointer to char
const char *const terry;//const pointer to char const
```

Various library function involving char pointer manipulation

```
str[n]cpy,str[n]cat,str[n]cmp,strdup,bcopy,memccopy,
memcpy,memmove,string,wcscpy,wcsncpy,index, rindex,
strcasecmp,strchr,strcmp, strcoll, strcspn,  strfry,
strlen,strncasecmp, strpbrk, strrchr,strsep, strspn,
strstr, strtok, strxfrm
```

- Dynamically allocating array, allocating on heap

```
- void *realloc(void *ptr, size_t size);// buffer remain uninitialized
int pia = malloc(sizeof(int)*10); // uninitialized

- void *realloc(void *ptr, size_t size);// initialized buffer with char 0;
int pia=calloc(sizeof(int)*10);

- void *realloc(void *ptr, size_t size);// re-allocate uninitialized buffer of new size if continuous memory is available for new size.
int pia=malloc(sizeof(int)*10);
pia=realloc(sizeof(int)*20);
```

If run on empty buffer malloc is called. 0 size free the memory

```
- void *realloc(void *ptr, size_t size);, frees the allocated memory
free(pia);
```

- Its legal to dynamically allocate empty array unless dereferenced

```
char arr[0]; // error
char arr = new char[0];// ok
char arr = new char[get_size()];//get_size() can return 0;
```

Day 1 Morning

3. Array Pointers References

- Definining and Initialzing Array
- Defining and initializing Pointers
- **Using Pointers to access array elements**
- Pointers and const qualifiers
- Dynamic allocated arrays
- Multidimensional array arugment to function

- Pointer works in dynamic allocation

- Array of pointers

```
classA *arrp=new classA[10];
```

- Pointer to array of size 10

```
classA (*ptoarr)=new classA[10];
classA arryofarry[3][10;
ptoarr=&arryofarry[2];

int (*pa)[10];
int *pi[10];

int arr[4][10]
pa=arr;
pa++


int (*pa)[10];
int arr[4][10];
pa=arr;
i=0;
for(j=0;j<4;j++){
for(k=0;k<10;k++)
arr[j][k]=i;
++i;
}
++pa;
```

```
for(j=0;j<10;j++)
cout<<pa[0][j];
```

Day 1 Morning

3. Array Pointers References

- Defining and Initialzing Array
- Defining and initializing Pointers
- Using Pointers to access array elements
- **Pointers and const qualifiers**
- Dynamic allocated arrays
- Multidimensional array arugment to function

read from the right to left.

```
const char *terry;//pointer to constant char
char *const terry;//const pointer to char
const char *const terry;//const pointer to char const
```

Day 1 Morning

3. Array Pointers References

- Defining and Initialzing Array
- Defining and initializing Pointers
- Using Pointers to access array elements
- Pointers and const qualifiers
- **Dynamic allocated arrays**
- Multidimensional array arugment to function

- malloc, calloc, realloc and free are used for dynamic allocation

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

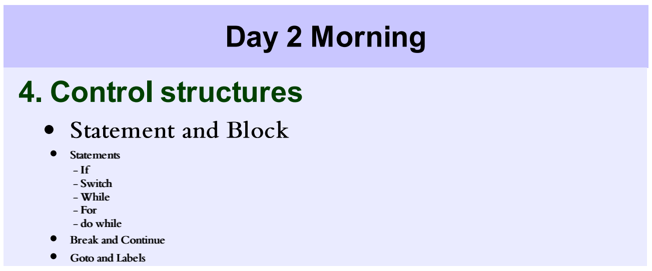
Day 1 Morning

3. Array Pointers References

- Defining and Initialzing Array
- Defining and initializing Pointers
- Using Pointers to access array elements
- Pointers and const qualifiers
- Dynamic allocated arrays
- **Multidimensional array arugment to function**

$ cat main.c

```
#include <stdio.h>

void printarray(int (*arr)[4]){
int i=0,j=0;
for (i=0;i<4;i++){
for (j=0;j<4;j++)
printf("%d ",arr[i][j]);
printf("
");
}
}
void preparearray(int (*arr)[4]){
int i=0,j=0;
for (i=0;i<4;i++)
for (j=0;j<4;j++)
arr[i][j]=i;
}

int main(int argc,char *argv[]){
int arr[4][4];
preparearray(arr);
printarray(arr);
return 0;
}

$ ./a.out
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3
```



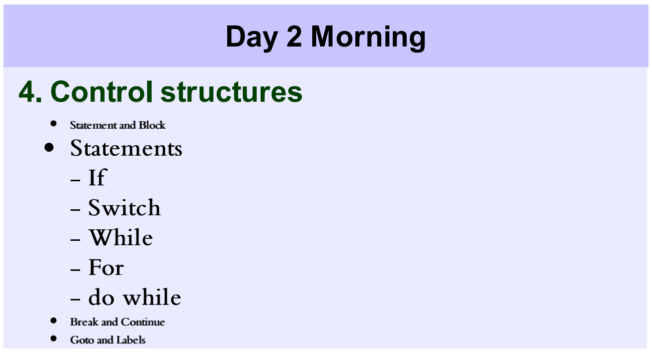
Day 2 Morning

4. Control structures

- Statement and Block
- Statements
  - If
  - Switch
  - While
  - For
  - do while
- Break and Continue
- Goto and Labels

A single statement is any valid string in C which ends with a semi colon.
e.g.
```
a = 6;
; // empty statement
printf ("I love C because...");
```

A compound statement is any number of single statements grouped together in curly braces. The curly braces do not end with a semi colon and stand in place of a single statement. Any pair of curly braces may contain local declarations after the opening brace. e.g.
Block statement
```
{
int i;
a = 6;
}

{
int a; // second definition but in different statement block
a = 6;
printf ("I love C because...");
}
```



Day 2 Morning

4. Control structures

- Statement and Block
- Statements
  - If
  - Switch
  - While
  - For
  - do while
- Break and Continue
- Goto and Labels

If
```
 if (<expression>) <statement> // simple form with no {}'s or else clause
 if (<expression>) { // simple form with {}'s to group statements
 <statement>
 <statement>
 }

if (<expression>) { // full then/else form
<statement>
}
else {
<statement>
}
```

Switch
The switch statement is a sort of specialized form of if used to efficiently separate different blocks of code based on the value of an integer.

```
switch (<expression>) {
case <const-expression-1>:
<statement>
break;
case <const-expression-2>:
<statement>
break;
case <const-expression-3>: // here we combine case 3 and 4
case <const-expression-4>:
<statement>
break;
default: // optional
<statement>
}
```

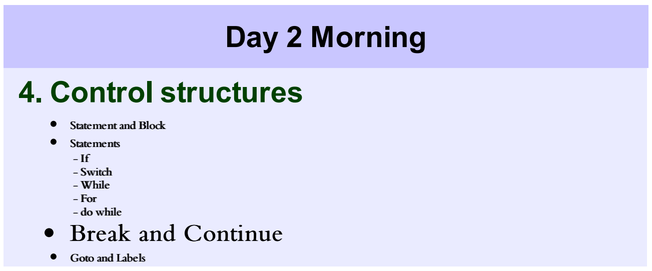
While Loop

The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires the parenthesis like the if.

```
while (<expression>) {
<statement>
}
```

Do-While Loop
Like a while, but with the test condition at the bottom of the loop. The loop body will always execute at least once. The do-while is an unpopular area of the language, most everyone tries to use the straight while if at all possible.

```
do {
<statement>
} while (<expression>)
```



Day 2 Morning

4. Control structures

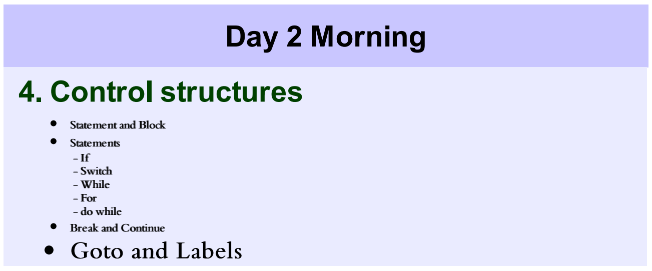
- Statement and Block
- Statements
  - If
  - Switch
  - While
  - For
  - do while
- Break and Continue
- Goto and Labels

Break
The break statement will move control outside a loop or switch statement.

```
while (<expression>) {
<statement>
<statement>
if (<condition which can only be evaluated here>)
break;
<statement>
<statement>
}
```

// control jumps down here on the break
Continue
The continue statement causes control to jump to the bottom of the loop, effectively skipping over any code below the continue.

```
while (<expression>) {
...
if (<condition>)
continue;
...
...
// control jumps here on the continue
}
```



Day 2 Morning

4. Control structures

- Statement and Block
- Statements
  - If
  - Switch
  - While
  - For
  - do while
- Break and Continue
- Goto and Labels

A goto statement in C programming language provides an unconditional jump from the goto to a labeled statement in the same function.
NOTE: Use of goto statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

Syntax
The syntax for a goto statement in C is as follows:

```
goto label;
..
.
label: statement;
```

Day 2 Morning

5. Function

- struct
  - Structures and functions
  - Array of structure
  - Self referential structure
  - Typedef
  - Unions
  - Bit-fields

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
A structure is user defined type.
All members are public.

```
struct fraction{
int numerator;
int denominator;
};
```

A struct declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct { ... } x, y, z;
```

is syntactically analogous to
```
int x, y, z;
```

struct initialization
A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct maxpt = { 320, 200 };
```

Structure as member

```
struct point{
int x;
int y;
};


struct rect{
struct point p1;
struct point p2;
};

struct {
int len;
char *str;
} *p;  //instantiation
```

then
```
++p->len
```
Increments len, not p, because the implied parenthesization is ++(p->len).

Structure and Function
 - pass by value
 - pass by address

```
struct addpoints(const struct point p1, const stuct point p2){
stuct point result;
...
return result;
}

struct addpoints(const struct point *p1,const struct point *p20){
...
return result;
}
```

Recursively defined structures
- when two structures refer to each other, one must be declared in
 incomplete(prototype) fashion

```
struct HUMAN;
struct PEN{
char name[NAME_LIMIT];
char species[NAME_LIMIT];
struct HUMAN *owner;
}fido={"Fido","Canis lupus familiaries"};
struct HUMAN{
char name[NAME_LIMIT];
struct PEN pets[PET_LIMIT];
}sam={"Sam",{fido}};
```

Array of Structures
Consider writing a program to count the occurrences of each C keyword.

```
struct key {
char *word;
int count;
} keytab[NKEYS];
```

Initialize

```
struct key {
char *word;
int count;
} keytab[] = {
"auto", 0,
"break", 0,
"case", 0,
"char", 0,
"const", 0,
"continue", 0,
"default", 0,
/* ... */
"unsigned", 0,
"void", 0,
"volatile", 0,
"while", 0
};
```

Typedefs

```
typedef struct{
} pstruct; //typedef
pstruct p;
```

- Self referential structure
Consider simple tree structure where each node has value and it pointer to left and right node of the tree.

```
struct tnode { /* the tree node: */
int data; /* number of occurrences */
struct tnode *left; /* left child */
struct tnode *right; /* right child */
};
```

Union
A union is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements

union-name.member
or
union-pointer->member

```
union u_tag {
int ival;
float fval;
char *sval;
} u;
```

The variable u will be large enough to hold the largest of the three types;

```
struct {
char *name;
int flags;
int utype;
union {
int ival;
float fval;
char *sval;
} u;
} symtab[NSYM];
```

the member ival is referred to as

```
symtab[i].u.ival
```

and the first character of the string sval by either of

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

- union can be used to find polarity of machine

```
union{
unsigned Int i;
char *c[4];
}un;
un.i=1
if(un.c[0])
printf("little endian
")
Else
printf("big endian");
```

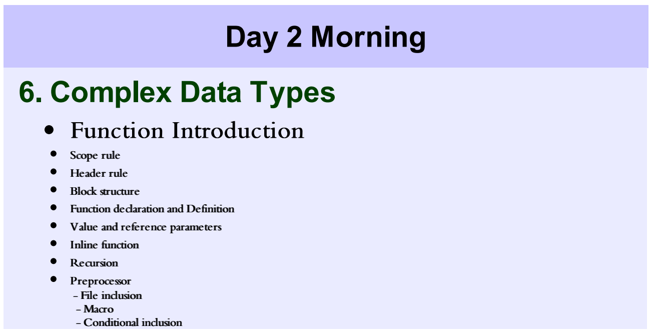
Bit fields
If space is a serious concern, select the number of bits used for each member
A bit-field, or field for short, is a set of adjacent bits within a single implementation-defined storage unit that we will call a "word."

```
struct  {
unsigned int is_keyword:1;
unsigned is_extern: 1;
unsigned is_static: 1;
}flags;

flags.is_extern = flags.is_static = 1; //turn on
flags.is_extern = flags.is_static = 0; // turn off
```

if (flags.is_extern == 0 && flags.is_static == 0) // to test them

Files are not arrays and they do not have addresses, so the &
operator cannot be applied on them.

Almost everything about fields is implementation-dependent. Whether a field may overlap a word boundary is implementation-defined. Fields need not be names; unnamed fields (a colon and width only) are used for padding. The special width 0 may be used to force alignment at the next word boundary.



Day 2 Morning

6. Complex Data Types

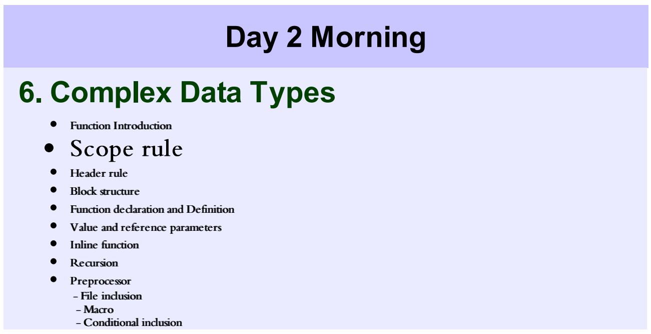
- Function Introduction
- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference parameters
- Inline function
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

Function
A small program(subroutine) that performs a particular task
- Input : parameter / argument
- Perform what ? : function body
- Output : return value

Modular programming design
- Large and complex task can be divided into smaller and simple task which is more easily solved (implemented).
- Also called
 - structured design
 - Top-down design
 - Divide-and-Conquer

/*
Computes double of a number.
Works by tripling the number, and then subtracting to get back to double.
*/

```
static int Twice(int num) {
int result = num * 3;
result = result - num;
return(result);
}
```



Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference parameters
- Inline function
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

The scope of a name is the part of the program within which the name can be used.
Global variable
- Declared, outside function block
- Accessible everywhere

- Global variable is destroyed only when a program is terminated
- Global variable is initialized before main function enters

Local variable (automatic variable?)
- Declared inside function body
- Accessible only in the function
- Local variable is created when a function is called and is destroyed when a function returns
- Storage allocation on stack.

Static variable (declared in a function)
- (Usually) accessible in the function
- Initialized to 0
- Goes to initialized or uninitialized section and not on stack
- Static variable persists until the program is terminated

```
main() { ... }
int sp = 0; // global variable, extern is allowed in other file
static sps=0; // extern is not allowed in other file
```

p24

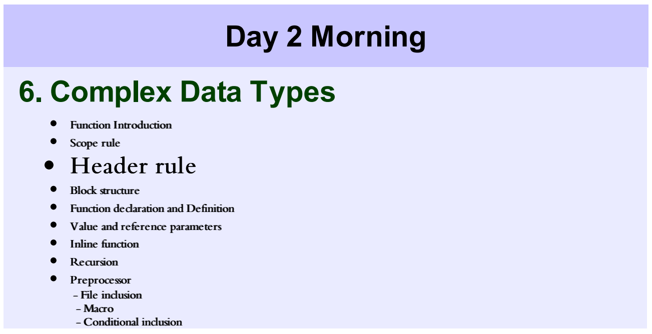
```
double val[MAXVAL];
void push(double f) { ... }
double pop(void) { ... }
```

$ cat test1.c

```
int sp=20;
```

$ cat main.c

```
#include <stdio.h>
extern int sp;
int main(){
printf("sp %d
",sp);
return 0;
}
```

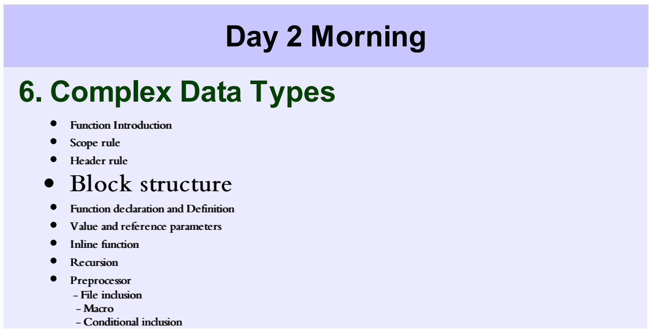


Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- **Header rule**
- Block structure
- Function declaration and Definition
- Value and reference parameters
- Inline function
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

A header file is a file with extension .h which contains C function declarations and macro definitions and to be shared between several source files.



Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- Header rule
- **Block structure**
- Function declaration and Definition
- Value and reference parameters
- Inline function
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

Variables can be defined in a block-structured fashion within a function. Declarations of variables (including initializations) may follow the left brace that introduces any compound statement, not just the one that begins a function. Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching right brace.

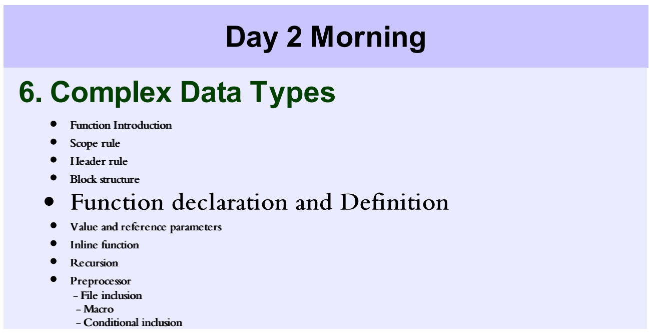
For example, in

```
if (n > 0) {
int i; /* declare a new i */
for (i = 0; i < n; i++)
...
}
```

or

simple block {

```
using namespace abc;
var=10;//var in abv
}
{
using namespace def;
var=20;//var in def
}
```



Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference parameters
- Inline function
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

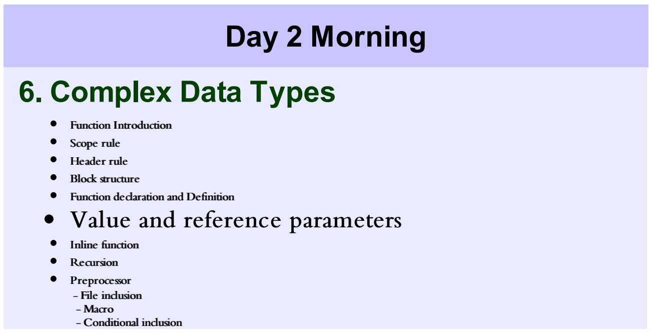
Function declaration does not have block.
```
void func();
extern funct();
```

Function declaration goes to header file.

Function definition has {} block.
Function when called callee function data is allocated on stack.
```
void func(){
}
```
Function definition goes to source file.

Function call stack
Supports the function call/return mechanism
- Each time a function calls another function, a stack frame (also known as an activation record) is pushed onto the stack
- Maintains the return address that the called function needs to return to the calling function
- Contains automatic variables - parameters and any local variables the function declares
- When the callee functions returns
 - Stack frame for the function call is popped
 - Control transfers to the return address in the popped stack frame

Stack overflow
- Due memory limitation new stack allocation is not possible



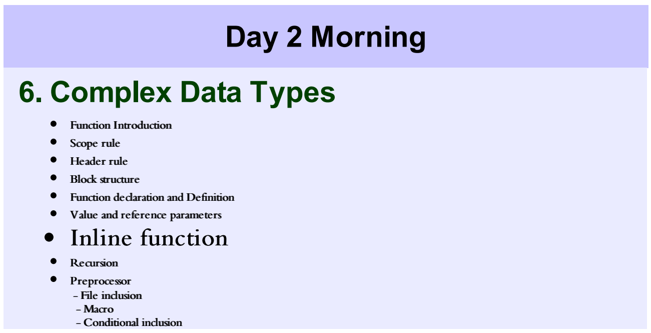
Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference parameters
- Inline function
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

Function caller passes argument to function callee as value. There is no reference in c.

```
int i=0;
int *pa=&i;
char *ps;
void func(int);
void func(int*);
void func(char **);

func(i);
func(pa);
func(&ps,10);

void func(char **pps,int isize){
*pps=malloc(isize*sizeof(char));
}
```



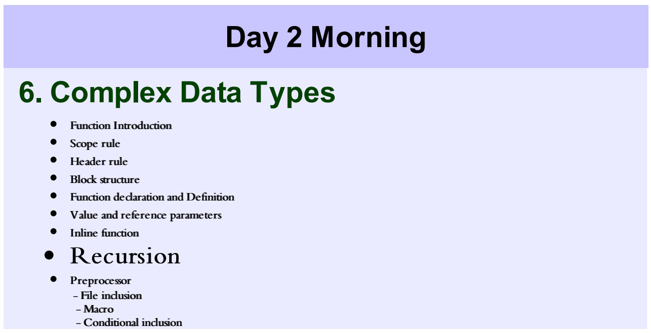
Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference parameters
- **Inline function**
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

The compiler will insert the complete body of the inline function in every place in the code where that function is used.
- Reduce overhead for function call & return
- Effective when a function is short and simple

```
inline int cube(int n){
return n*n*n;
}
```



Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference parameters
- Inline function
- **Recursion**
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

C functions may be used recursively; that is, a function may call itself either directly or indirectly.

```
#include <stdio.h>
/* printd: print n in decimal */
void printd(int n)
{
if (n < 0) {
putchar('-');
n = -n;
}
if (n / 10)
printd(n / 10);
putchar(n % 10 +'0');
}

--------------
void printd(int 12)
{
if (n < 0) {
putchar('-');
n = -n;
}
if (n / 10)
printd(n / 10);      -----------------
putchar(n % 10 +'0');                 |
}                                     |
-------------                         |
-------------                         |
void printd(int 1) <----------------
{
if (n < 0) {
putchar('-');
n = -n;
}
if (n / 10)
printd(n / 10);
putchar(n % 10 +'0');
}
-------------
```

Day 2 Morning

6. Complex Data Types

- Function Introduction
- Scope rule
- Header rule
- Block structure
- Function declaration and Definition
- Value and reference parameters
- Inline function
- Recursion
- Preprocessor
  - File inclusion
  - Macro
  - Conditional inclusion

File inclusion

File inclusion makes it easy to handle collections of #defines and declarations (among other things). Any source line of the form

```
#include "filename"
```
or
```
#include <filename>
```

Macro

A definition has the form

```
#define name replacement text
#define paste(front, back) front ## back
```

Conditional Inclusion

Conditional inclusion provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

```
#if SYSTEM == SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

Day 3 Morning

7. Input and Output

- **Standard Input and Output**
- Buffered i/o
- Variable length argument list
- File access
- Line input and output
- Error handling – stderr and exit

Standard Input
Input from system console

```
int getchar(void)
prog <infile   //prog is user program with getchar
anotherprog | prog
```

Output to system console
```
int putchar(int)
prog >outfile
prog | anotherprog
```

Various unbuffered APIs
```
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
int close(int fd);
```

Day 3 Morning

7. Input and Output

- Standard Input and Output
- **Buffered i/o**
- Variable length argument list
- File access
- Line input and output
- Error handling – stderr and exit

```
          --------------         --------
          | User Space | <-----> |Buffer|
          --------------         --------
                ^
                |
                v
--------------------------------------
              Kernel
```

```
 FILE *fopen(const char *path, const char *mode);
 FILE *fdopen(int fd, const char *mode);
 int fprintf(FILE *stream, const char *format, ...);
 int fgetc(FILE *stream);
 int fputc(int c, FILE *stream);
 int fclose(FILE *fp);
```

Day 3 Morning

7. Input and Output

- Standard Input and Output
- Buffered i/o
- **Variable length argument list**
- File access
- Line input and output
- Error handling – stderr and exit

```
#include <stdarg.h>
double average(int num,...)
{
va_list valist;
double sum = 0.0;
int i;
/* initialize valist for num number of arguments */
va_start(valist, num);
/* access all the arguments assigned to valist */
for (i = 0; i < num; i++)
{
sum += va_arg(valist, int);
}
/* clean memory reserved for valist */
va_end(valist);
return sum/num;
}
```

Define a function with last parameter as ellipses and the one just before the ellipses is always an int which will represent number of arguments.

Create a va_list type variable in the function definition. This type is defined in stdarg.h header file.

Use int parameter and va_start macro to initialize the va_list variable to an argument list. The macro va_start is defined in stdarg.h header file.

Use va_arg macro and va_list variable to access each item in argument list.

Use a macro va_end to clean up the memory assigned to va_list variable.

Day 3 Morning

7. Input and Output

- Standard Input and Output
- Buffered i/o
- Variable length argument list
- File access
- Line input and output
- Error handling - stderr and exit

```
FILE *fp;
FILE *fopen(char *name, char *mode);
int getc(FILE *fp)
```

getc returns the next character from the stream referred to by fp;
it returns EOF for end of file or error.

```
int putc(int c, FILE *fp)
```

putc writes the character c to the file fp and returns the character written, or EOF if an error occurs. Like getchar and putchar, getc and putc may be macros instead of functions.

getchar and putchar can be defined in terms of getc, putc, stdin, and stdout as follows:

```
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
```

For formatted input or output of files, the functions fscanf and fprintf may be used. These are identical to scanf and printf, except that the first argument is a file pointer that specifies the file to be read or written; the format string is the second argument.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)

#include <stdio.h>
/* cat: concatenate files, version 1 */
main(int argc, char *argv[])
{
FILE *fp;
void filecopy(FILE *, FILE *);
if (argc == 1) /* no args; copy standard input */
filecopy(stdin, stdout);
else
while(--argc > 0)
if ((fp = fopen(*++argv, "r")) == NULL) {
printf("cat: can't open %s", *argv);
return 1;
} else {
filecopy(fp, stdout);
fclose(fp);
}
return 0;
```

Day 3 Morning

7. Input and Output

- Standard Input and Output
- Buffered i/o
- Variable length argument list
- File access
- Line input and output
- Error handling - stderr and exit

The standard library provides an input and output routine fgets that is similar to the getline function that we have used in earlier chapters: char *fgets(char *line, int maxline, FILE *fp)

fgets reads the next input line (including the newline) from file fp into the character array line; at most maxline-1 characters will be read. The resulting line is terminated with '\0'. Normally fgets returns line; on end of file or error it returns NULL. (Our getline returns the line length, which is a more useful value; zero means end of file.) For output, the function fputs writes a string (which need not contain a newline) to a file:
```
int fputs(char *line, FILE *fp)
```

It returns EOF if an error occurs, and non-negative otherwise.
The library functions gets and puts are similar to fgets and fputs, but operate on stdin and stdout. Confusingly, gets deletes the terminating '
', and puts adds it.

To show that there is nothing special about functions like fgets and fputs, here they are, copied from the standard library on our system:

```
/* fgets: get at most n chars from iop */
char *fgets(char *s, int n, FILE *iop)
{
register int c;
register char *cs;
cs = s;
while (--n > 0 && (c = getc(iop)) != EOF)
if ((*cs++ = c) == ' ')
break;
*cs = '\0';
return (c == EOF && cs == s) ? NULL : s;
}
/* fputs: put string s on file iop */
int fputs(char *s, FILE *iop)
{
int c;
while (c = *s++)
putc(c, iop);
return ferror(iop) ? EOF : 0;
}
```

For no obvious reason, the standard specifies different return values for ferror and fputs. It is easy to implement our getline from fgets:

```
/* getline: read a line, return length */
int getline(char *line, int max)
{
if (fgets(line, max, stdin) == NULL)
return 0;
else
return strlen(line);
}
```

Day 3 Morning

7. Input and Output

- Standard Input and Output
- Buffered i/o
- Variable length argument list
- File access
- Line input and output
- Error handling – stderr and exit

a)Program exit
A program can exit with exit(int) system call. program return status can be checked later. If program crashes because of signal then system fills the return status. Generally 0 is success and nonzero is failure. $? shell command prints the exit status of last run program

```
$ ls /
bin coda etc lib misc nfs proc sbin usr
boot dev home lost+found mnt opt root tmp var

$ echo $?
0

$ ls bogusfile
ls: bogusfile: No such file or directory
```

```
$ echo $?
1
```

b) System call failure
Global variable errno carries the last error encountered in any system call. Various constants i.e EINTR, EROFS etc are defined with particular error number and errno variable need to be checked against these constants rather then integer. stderror(int) function prints the error description with errno passed as an argument.

Day 3 Morning

8. Storage class Specifier

- Automatic
- Const
- Global
- Extern
- Register
- Static
- Volatile

The auto storage class is the default storage class for all local variables.

```
{
int mount;
auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

Day 3 Morning

8. Storage class Specifier

- Automatic
- Const
- Global
- Extern
- Register
- Static
- Volatile

The purpose of const is to announce objects that may be placed in read-only memory, and perhaps to increase opportunities for optimization.

Day 3 Morning

8. Storage class Specifier

- Automatic
- Const
- Global
- Extern
- Register
- Static
- Volatile

Variable defined in global scope of the file.
Global variables are visible across the program.
File using the global variable need to declare it as extern in order to that file compiled.

Day 3 Morning

8. Storage class Specifier

- Automatic
- Const
- Global
- Extern
- Register
- Static
- Volatile

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files.

```
#include <stdio.h>
int count ; //definition
extern void write_extern();  //defined somewhere else

main()
{
write_extern();
}
Second File: write.c
#include <stdio.h>
extern int count; //only declaration, defined somewhere else
void write_extern(void)
{
count = 5;
printf("count is %d
", count);
```

Day 3 Morning

8. Storage class Specifier

- Automatic
- Const
- Global
- Extern
- **Register**
- Static
- Volatile

Register declared are opted to be put in registers.

```
register int x;
register char c;
```

The register declaration can only be applied to automatic variables and to the formal parameters of a function.

```
f(register unsigned m, register long n)
{
register int i;
...
}
```

It is not possible to take the address of a register variable.

Day 3 Morning

8. Storage class Specifier

- Automatic
- Const
- Global
- Extern
- Register
- **Static**
- Volatile

The static declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled.

<<a.c>>
```
static int a; // file scoped
void function(){
static int a; // function scoped
}
```

Day 3 Morning

8. Storage class Specifier

- Automatic
- Const
- Global
- Extern
- Register
- Static
- **Volatile**

The purpose of volatile is to force an implementation to suppress optimization that could otherwise occur. For example, for a machine with memory-mapped input/output, a pointer to a device register might be declared as a pointer to volatile, in order to prevent the compiler from removing apparently redundant references through the pointer.

Except that it should diagnose explicit attempts to change const objects, a compiler may ignore these qualifiers.

```
Port
---------------------------------------------
| control register |     |  data register   |
---------------------------------------------


typedef short int control;
typedef short int data;
#define ENABLE 0x40
#define READY 0x80
typedef struct port port;
struct port
{
control c;
data d;
};
typedef struct ioport ioport;
struct ioport
{
port in, out;
};

//Using the declarations above, assign one memory mapped address to it
ioport *const pio = (ioport *)0xFF70;

// check if control register ready bit is ready for data to be written
while (pio->out.c & READY == 0)
;

//write carriage return
pio->out.d = '\r';
while (pio->out.c & READY == 0)
;
pio->out.d = '
';

//.Compiler start doing the optimization in following ways.
//Optimization

if (pio->out.c & READY == 0)
for (;;)
;
pio->out.d = '\r';
if (pio->out.c & READY == 0)
for (;;)
;
pio->out.d = '
';
Optimization
if (pio->out.c & READY == 0)
for (;;)
;
pio->out.d = '\r';
pio->out.d = '
';

//Further Optimization where '\r' is removed as redundant
if (pio->out.c & READY == 0)
for (;;)
;
pio->out.d = '
';


Solution
ioport volatile *const pio
= (ioport *)0xFFA0;
```

Day 3 Morning

9. Misc

- Debugging with gdb
- 64 bit data i/o

The GNU Debugger (GDB) is used to stop through code, set breakpoints and examine the value of local variables.

Makefile for simple main.c in order to have a binary.
$ cat Makefile

```
CFLAGS=-g
all: main.o
        gcc $(CFLAGS) -o main main.o
main.o: main.c
        gcc $(CFLAGS) -c main.c -o main.o
clean:
        rm -f main.o main
```

[sc@localhost ~]$ gdb main

```
GNU gdb Red Hat Linux (6.5-16.el5rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...Using host libthread_db library "/lib64/libthread_db.so.1".
(gdb) break func
Breakpoint 1 at 0x400484: file main.c, line 6.
(gdb) run
Starting program: /home/sc/main
Breakpoint 1, func (si=0x7fff2dfdb6c0) at main.c:6
6       si->i=10;
(gdb) x /4xb 0x7fff2dfdb6c0
0x7fff2dfdb6c0: 0x14    0x00    0x00    0x00
(gdb) bt
#0  func (si=0x7fff2dfdb6c0) at main.c:6
#1  0x00000000004004c4 in main (argc=1, argv=0x7fff2dfdb7b8) at main.c:12
(gdb) l
1       #include <stdio.h>
2       struct samples{
3       int i;
4       };
5       void func(struct samples *si){
6       si->i=10;
7       printf("%d
",si->i);
8       }
9       int main(int argc,char *argv[]){
10      struct samples ss;

(gdb) n
7       printf("%d
",si->i);
(gdb) x /4xb 0x7fff2dfdb6c0
0x7fff2dfdb6c0: 0x0a    0x00    0x00    0x00
(gdb) p si->i
$1 = 10
```

Day 3 Morning

9. Misc

- Debugging with gdb
- 64 bit data i/o

```
d - input of signed decimal integer
i - input of a signed integer value
u - input of an unsigned decimal
o - unsigned octal integer
x - input of an unsigned hexadecimal

#include <stdio.h>
#include <inttypes.h>
int main(){
uint64_t uval=0x1234567890abcdef;
int64_t n=0xff;
printf("%"PRIu64"
", uval);
printf("%"PRId64"
",n);
return 0;
}
```

```
$ ./a.out
1311768467294899695
255
```

p37