C Programming Essentials

Unit 2: Structured Programming

CHAPTER 6: STRUCTURED PROGRAMMING (FUNCTIONS)

DR. ERIC CHOU IEEE SENIOR MEMBER

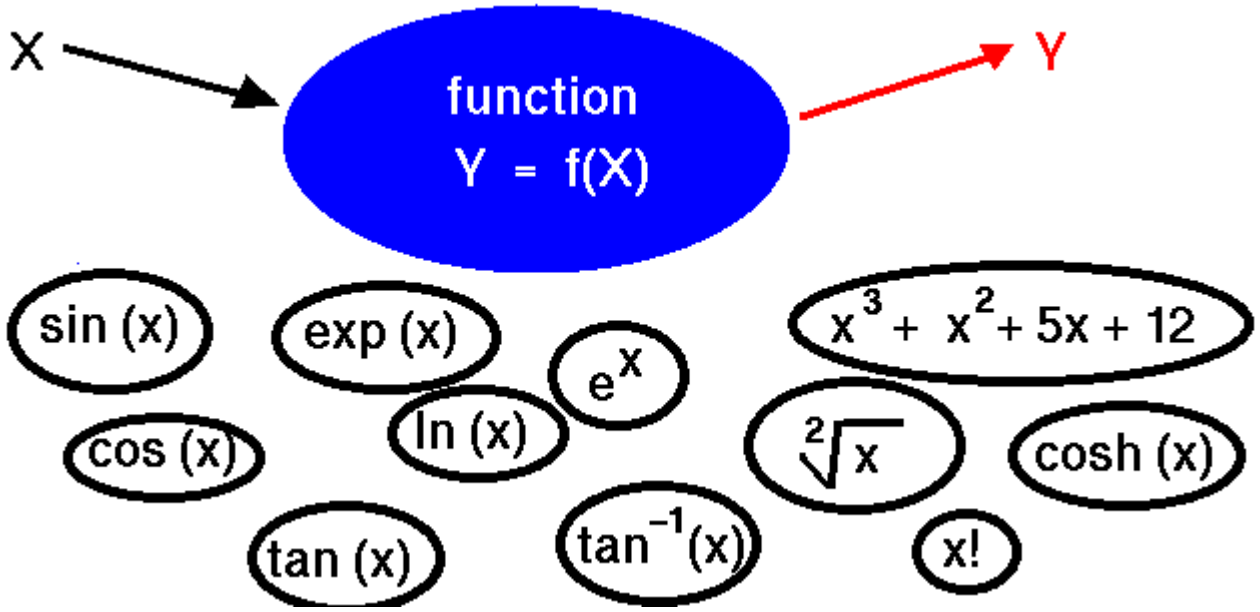
LECTURE 1

Basic Functions (I)

Why functions?

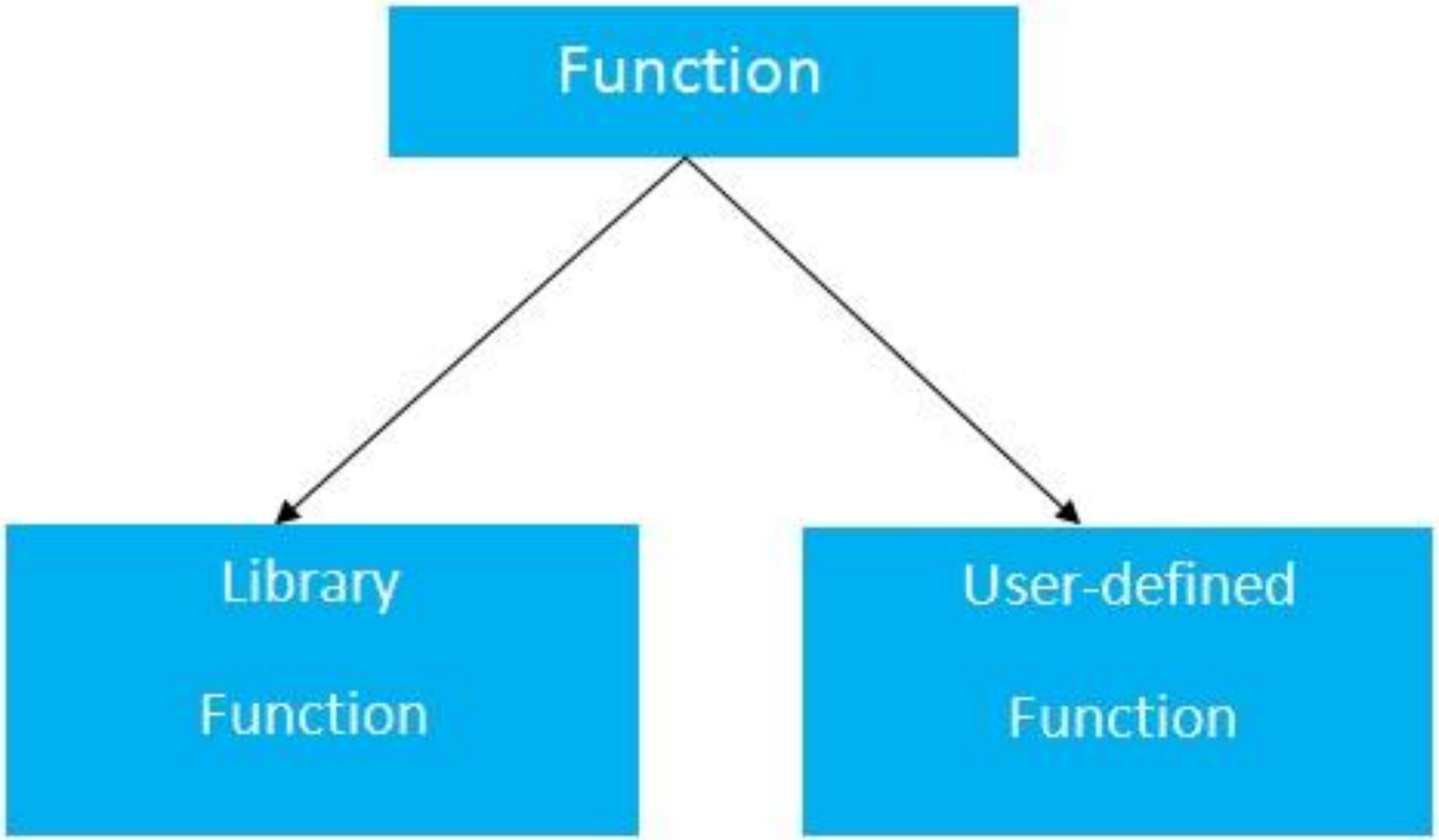
- Break longer jobs into conceptually smaller jobs which are precisely defined.
- C program generally consists of many small functions.
- A piece of a code which repeats at many places can be written only once and used again and again.
- Debugging and maintenance is easy.

Mathematical Functions and C Functions



Functions:

- Perform a specific task.
- Modularization
- Reusability
- Abstraction
- Return some result (optional in C, required in mathematical functions)


Function
Library
Function
User-defined
Function

function

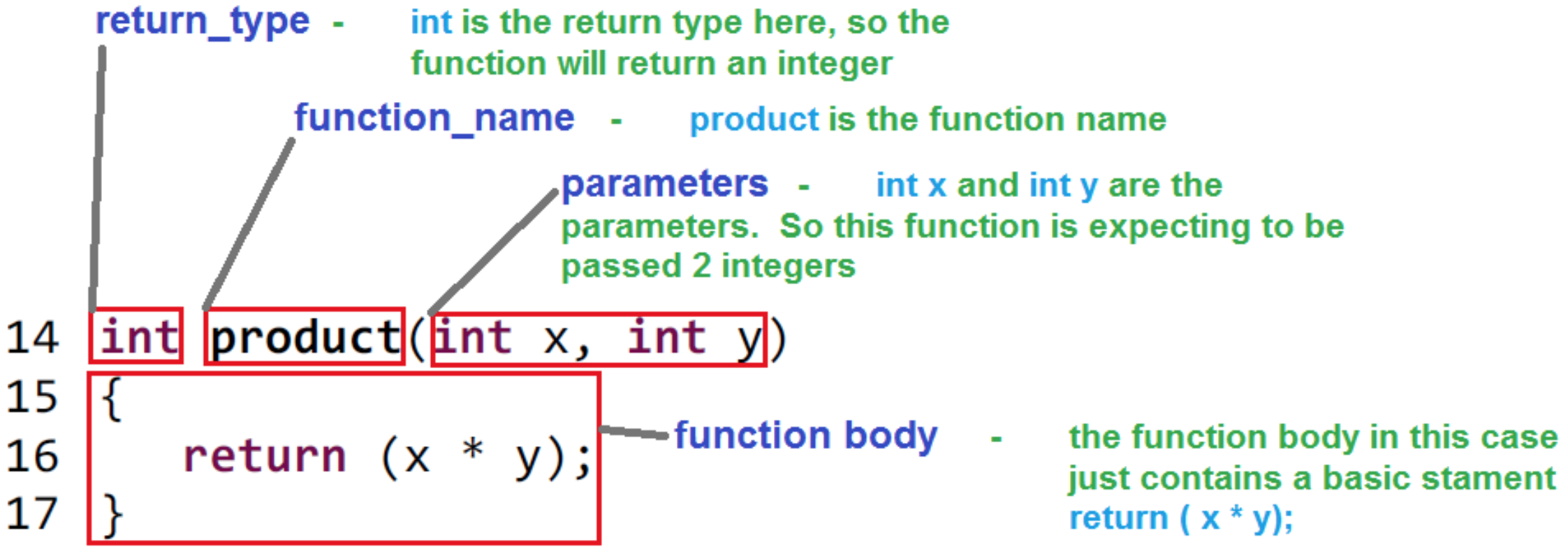
Definition: A block of code which performs some well defined computational task.

- a function may have some input
- a function may also have some output

Example: *int add(int , int)*

Note: Can visualize as a mathematical function which takes a value and returns another value.

$$f(x, y) = x * y$$


return_type - int is the return type here, so the function will return an integer
function_name - product is the function name
parameters - int x and int y are the parameters. So this function is expecting to be passed 2 integers
14 int product(int x, int y)
15 {
16 return (x * y);
17 }
function body - the function body in this case just contains a basic stament return (x * y);

Terminologies

Function Declaration/ Prototype (header):

return_type function_name(argument_types);

```
int func(int , int);
```

Function Definition:

```
int func ( int a, int b){
  printf("Welcome to func");
  return (a + b);
}
```

Terminologies

- The value passed to a method is called its
  “argument” (actual arguments)
- The variable which receives the arguments is called
  “parameter” (formal arguments)
- Parameters are declared inside the parenthesis and we must declare the type of the parameter.

- Argument may be an expression but,
- *type* (argument) = *type* (parameter)
- Call-by-value is used for C language.

Functional Declaration and Definition

style1.c style2.c style3.c

Go gcc!!!

3 ways to declare functions

style1.c

```c
#include <stdio.h>
void add_print(int , int); //function declaration
int main(){
   int a=4;
   int b=5;
   printf("Entering 'add_print' function\n");
   add_print(a,b);
   printf("Just came from 'add_print' function\n");
return 0;
}
//function definition
void add_print(int val1,int val2){
   int c;
   printf("The two values entered are:%d,%d \n",val1,val2);
   c=val1+val2;
   printf("Sum of numbers entered is:%d \n",c);
}
```

style2.c

```c
#include <stdio.h>
//function definition
void add_print(int val1,int val2){
   int c;
   printf("The two values entered are:%d,%d \n",val1,val2);
   c=val1+val2;
   printf("Sum of numbers entered is:%d \n",c);
}
int main(){
   int a=4;
   int b=5;
   printf("Entering 'add_print' function\n");
   add_print(a,b);
   printf("Just came from 'add_print' function\n");
   return 0;
}
```

style3.h

```c
void add_print(int, int);
```

style3.c

```c
#include <stdio.h>
#include "style3.h"

int main(){
   int a=4;
   int b=5;
   printf("Entering 'add_print' function\n");
   add_print(a,b);
   printf("Just came from 'add_print' function\n");
return 0;
}
//function definition
void add_print(int val1,int val2){
   int c;
   printf("The two values entered are:%d,%d \n",val1,val2);
   c=val1+val2;
   printf("Sum of numbers entered is:%d \n",c);
}
```

LECTURE 2

Basic Functions (II)

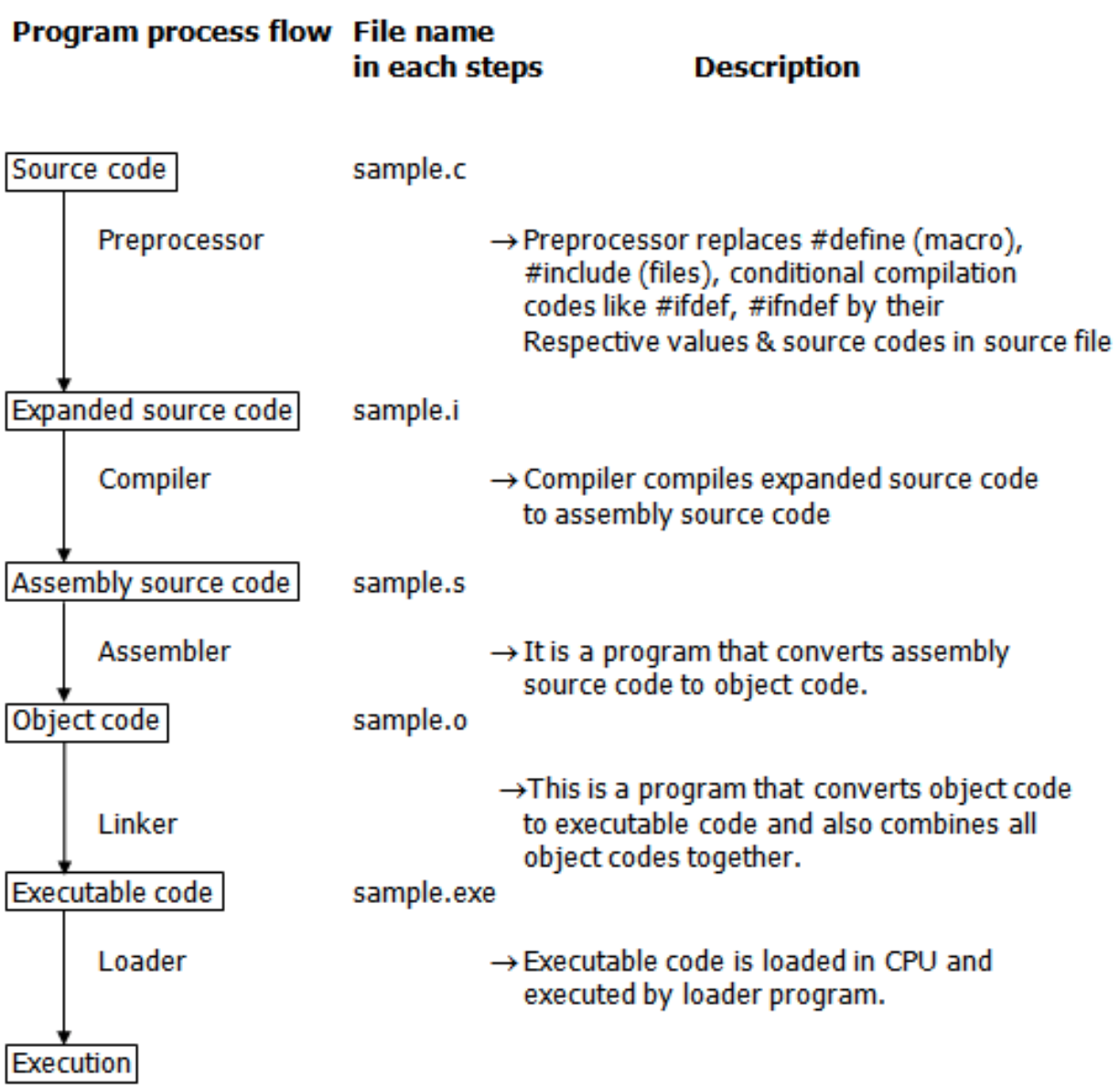
Issues on Functions

Always use function prototypes (put them in .h file)

```
int myfunc (char *, int, struct MyStruct *);
int myfunc_noargs (void);
void myfunc_noreturn (int i);
```

C and C++ are *call by value*, copy of parameter passed to function

- C++ permits you to specify pass by reference
- if you want to alter the parameter then pass a pointer to it (or use references in C++)


Program process flow
File name in each steps
Description
Source code
sample.c
Preprocessor
→ Preprocessor replaces #define (macro), #include (files), conditional compilation codes like #ifdef, #ifndef by their Respective values & source codes in source file
Expanded source code
sample.i
Compiler
→ Compiler compiles expanded source code to assembly source code
Assembly source code
sample.s
Assembler
→ It is a program that converts assembly source code to object code.
Object code
sample.o
Linker
→ This is a program that converts object code to executable code and also combines all object codes together.
Executable code
sample.exe
Loader
→ Executable code is loaded in CPU and executed by loader program.
Execution

In-line Function

If performance is an issue then use inline functions, generally better and safer than using a macro. Common convention

- define prototype and function in header or **name.h** file
- static inline int myinfunc (int i, int j);
- static inline int myinfunc (int i, int j) { ... }

Inline Function in C

- An **inline function** is a combination of macro & function. At the time of declaration or definition, function name is preceded by word **inline**.
- Function calls involve **execution-time** overhead.
- Inline functions to help **reduce** function **call overhead**, especially for small functions
- When inline functions area used, the overhead of function call is **eliminated**. Instead, the executable statements of the function are copied at the place of each function call. This is done by the compiler.

Inline Function in C

- Inline Function is powerful concept in C programming language. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.
- To make any function inline function just preceded that function with inline keyword.
- **Why use Inline function**
- Whenever we call any function many time then, it take a lot of extra time in execution of series of instructions such as saving the register, pushing arguments, returning to calling function. For solve this problem in C introduce inline function.
- **Advantage of Inline Function**
- The main advantage of inline function is it make the program faster.

Inline Function in C

- **Inline functions provide following advantages:**
  - Function call overhead doesn't occur.
  - It speeds up your program by avoiding function calling overhead.
  - It save overhead of return call from a function.

Demo Program: in.c

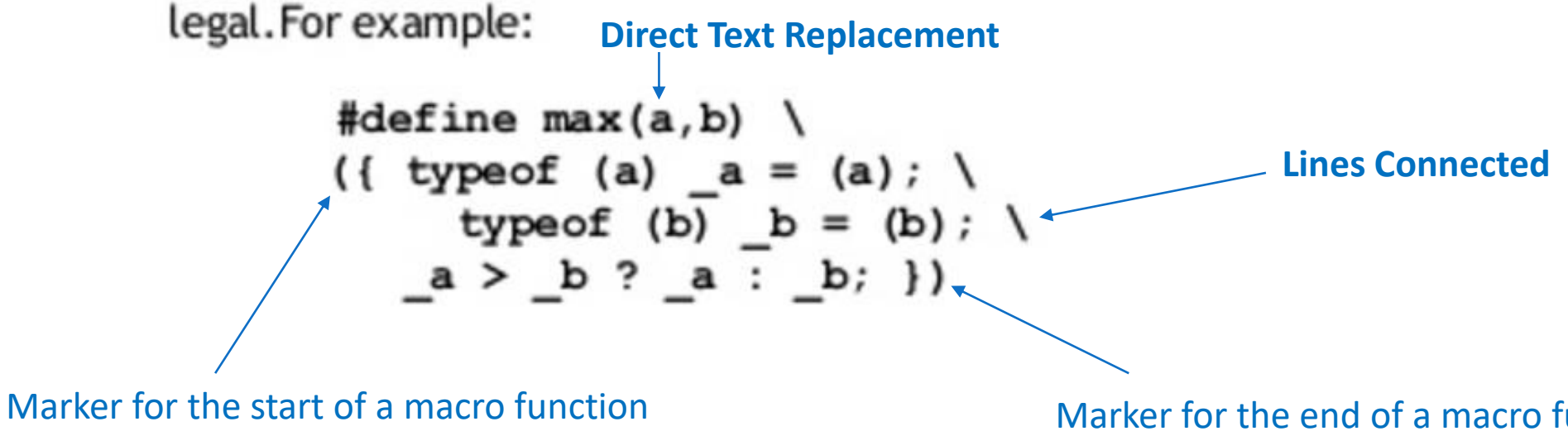
Go gcc!!!

```c
#include <stdio.h>
static inline int max(a, b) {return  (a>=b) ? a: b;}

int main(void){
    int x = 1;
    int y = 2;

    int c = max(x, y);

    printf("%d %d %d\n", x, y, c);
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\exmaples>gcc in.c -o in

C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\exmaples>in
1 2 2

C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\exmaples>
```

C Macro Function

If a **#define** directive does not fit on a single line, it can be continued on subsequent lines. All lines of the directive except the last line must end with a backslash(\) character. A directive can be split only at a point where a space is legal. For example:



Macro Function

- **Macros and functions may behave differently if an argument is referenced multiple times:**
  - a function argument is evaluated once, before the call
  - a macro argument is evaluated each time it is encountered in the macro body.
- **Example:**

```
int dbl(x) { return x + x;}
...
u = 10; v = dbl(u++);
printf("u = %d, v = %d", u, v);

    prints: u = 11, v = 20
```

```
#define  Dbl(x)   x + x
...
u = 10; v = Dbl(u++);
printf("u = %d, v = %d", u, v);

    prints: u = 12, v = 21
```

Dbl(u++)
expands to:
u++ + u++

Demo Program:

mac.c

Go gcc!!!

```c
#include <stdio.h>
#define max(a, b)                       \
        (                               \
            { typeof(a) _a = (a); \
              typeof(b) _b = (b); \
              _a > _b ? _a: _b;} \
         )

int main(void){
    int x = 1;
    int y = 2;

    int c = max(x, y);

    printf("%d %d %d\n", x, y, c);
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\exmaples>gcc mac.c -o mac

C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\exmaples>mac
1 2 2
```

	inline function	macro
1	These are functions provided by C++	Macros are preprocessor directives.
2	Inline keyword is used to declare the function as inline.	#define is used to declare the macro.
3	It can be define inside or outside the class.	It cannot be declare inside the class.
4	Inline functions are parsed by the compiler.	Macros are expanded by the C++ preprocessor.
5	Inline function can access the data member of the class	Macros cannot access the data member of the class
6	compiler replaces the function call with the function code	C preprocessor replaces every occurrence of macro template with its corresponding definition.
7	Inline functions follows strict parameter type checking	Macros does not follows parameter type checking
8	Inline functions may or may not be expanded by the compiler. Its depends upon the compiler's decision whether to expand the function inline or not.	Macros are always expanded.
9	Can be used for debugging a program	Cannot be used for debugging as they are expanded at pre-compile time.

10	inline int sum(int a, int b)
	{
	return (a+b);
	}
		#define SUM(a,b) (a+b)

LECTURE 3

Basic Functions (III)

Function as a Task

WELCOME()

Functions: Example welcome.c

Perform a task.

```
#include <stdio.h>

void welcome(){
  printf("welcome. \n");
}

int main(void){
   welcome(); welcome(); welcome();
}
```

Function as a Mathematical Function

SIN()

Functions: Example sin.c

Sine Function

- Sine function is included in the standard library and is declared in math.h
- The example here uses Taylor series expansion. The successive terms are added till the term becomes smaller than the machine epsilon.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \dots$$

```
#include <limits.h>
double sin(double x){
  double sinValue = 0.0;
  double xSquared;
  double term;
  int n = 1;
  xSquared = x * x;
  term = x;
  while ( fabs(term) > DBL_EPSILON ) {
    sinValue += term;
    n += 2;
    term *= (- xSquared)/(n - 1)/n;
  }
  return sinValue;
}
```

Function as a Logical Condition

ISPRIME()

Functions: Example

Prime Numbers

- Print a table as follows

```
1*   2*   3*   4    5* … 10
11* 12   13* 14   15 … 20
```

  up to 100. All primes are starred.
- The main function prints the nice looking table. And it is not about finding primes.
- The functinon isPrime(n) is expected to return 1(true) if n is a prime and 0(false) otherwise.

```
#include <stdio.h>
int main(void){
  int i;
  for( i = 1; i <= 100; i++ ) {
    prinf("%d", i);
    if ( isPrime(i) )
      printf("*");
    if ( i % 10 == 0 )
      printf("\n");
    else
      printf("\t");
  }
  return 0;
}
```

Functions: Example

Prime Numbers

- A rather simple implementation of isPrime function is given here.
- The function has same structure as main function we saw before, except that int before the name of the function and return statement.

```
int isPrime(int num){
  int  i;
  if ( num < 3 ) return 1;
  for (i=2; i<=(int) sqrt(num);i++ )
    if ( num % i == 0 ) break;
 if ( num % i ) return 1;
 return 0;
}
```

Function for Update of Value

NEWI() NEWI2()

Functions

- The syntax of a function is

```
return-type function-name(argument-list)
{
   Declarations and statements
}
```

- **Smallest Function** is

```
emptyFunction() {}
```

- Rules for the function name are same as that of the variables.
- Return-type must be one of valid data-types or `void`. If it is not mentioned then `int` is assumed. It is not mandatory to return a value. The calling program is free to ignore the return value.
- The form of argument list is

```
type1 arg1, type2 arg2,..., typen argn
```

- it is understood that the arguments have been declared.

Arguments

- The term **argument** (actual argument, actual parameter) is used for the variables that are passed to the function by a calling program. The term parameter is used for the variables received by the function as described in the function declaration.
- The arguments are passed by value. This means a separate copy of the variable is passed to the function. The values of the variables in the calling program are not affected by the changes to the parameters in the function definition.
- The types of the arguments passed and the parameters declared must match. If they don’t match, automatic type casting is applied. If these still do not match, there is an error.

Arguments: new.c

- This program will output three lines:

```
4
5
4
5
5
```

```
#include <stdio.h>
newI (int I) {
 I = 5;
 printf("%d\n", I);
}
newI2 (int *I) {
 *I = 5;
 printf("%d\n", *I);
}

int main(void) {
 int    I;
 I = 4;
 printf("%d\n", I);
 newI(I);
 printf("%d\n", I);
 newI2(&I);
 printf("%d\n", I);
 return 0;
}
```

Function Creator of Objects

NEWRECORD()

Creation of Objects: newRecord.c

- This program will output one lines:

```
0 0
```

```
#include <stdio.h>
#include <stdlib.h>
typedef struct a { int x;  int y;}
 record;

record *newRecord(void){
 record *r=calloc(1, sizeof(record));
 r->x =0; r->y=0;
 return r;
}

int main(void){
  record *r = newRecord();
  printf("%d %d\n", r->x, r->y);
  return 0;
}
```

LECTURE 4

Break Levels (II)

Break Level

- **{}** no operation or pass // {return 0;} if return needed
- **continue:** bypass the rest of a iteration.
- **break:** bypass the rest of a loop or switch statement
- **exit(0):** quit the whole program (exit with system error code)

Demo Program:

pbreak.c

Go gcc!!!

If statement is skipped →

```c
#include <stdio.h>

int main(){
  int i=0;
  for (i=0; i<5; i++){
     if (i==3) { /*pass*/ };
     printf("%d\n", i);
  }
  printf("End\n");
  return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>gcc pbreak.c -o pbreak

C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>pbreak
0
1
2
3
4
End
```

Demo Program:

cbreak.c

Go gcc!!!

```c
#include <stdio.h>

int main(){
  int i=0;
  for (i=0; i<5; i++){
    if (i==3) continue;
    printf("%d\n", i);
  }
 printf("End\n");
 return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>gcc cbreak.c -o chreak

C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>cbreak
0
1
2
4
```

End show up and skip 3 →

Demo Program:

bbreak.c

Go gcc!!!

```c
#include <stdio.h>

int main(){
  int i=0;
  for (i=0; i<5; i++){
    if (i==3) break;
    printf("%d\n", i);
  }
 printf("End\n");
 return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>gcc bbreak.c -o bbreak

C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>bbreak
0
1
2
End
```

End show up and break at 3 →

Demo Program:

ebreak.c

Go gcc!!!

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

int main(){
  int i=0;
  for (i=0; i<5; i++){
    if (i==3) exit(0);
    printf("%d\n", i);
  }
  printf("End\n");
  return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>gcc ebreak.c -o ebreak

C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\breaks>ebreak
0
1
2
```

End doesn't show up

LECTURE 5

Header File

Functional Declaration Line in C

- A function declaration introduces the function name and its type.
- A function definition associates the function name/type with the function body.
- The functional declarations can be in the same file as the function definition. They can also be put into different files.
- The functional declaration is used to inform the compiler the size of the memory block that needed when the function is called and returned. (Call frame push and pop in the call stack.)
- Functional declaration can be put into a header file. It can be included for any module that call those functions.
- The functional definition can be put in the source file (.c, .cpp) files)

Function Prototype

Allows for a function to be used before it is defined:

```
double func(double, double); //prototype

main(void){
  double result = func(10,20);
}

double func(double x, double y){
  return x * y;
}
```

Source and Header Files

- Just as in C, place related code within the same module (i.e. file).
- Header files (*.h) export interface definitions
  - Function prototypes, data types, macros, inline functions and other common declarations.
- Do not place source code (i.e. definitions) in the header file with a few exceptions.
  - inline'd code
  - Class definitions (C++)
  - const definitions
- C Preprocessor is used to insert common definitions (in .h file) into source file before compilation.

```c
//Avoidance of re-declaration
#ifndef _HUFFMAN_H
#define _HUFFMAN_H
// Inclusion of other .h files (from public library or user defined files)
#include <stdint.h>
#include <stdbool.h>
#include "code.h"
// definition of constants
#ifndef NIL
#define NIL (void *) 0
#endif
// definition of macros
#define DEBUG 1
// type declarations
typedef struct DAH{
  uint8_t symbol ;
  bool leaf ;
  uint64_t count ;
  treeNode *left , *right ;
};
// inline functions
static inline void delNode(treeNode *h) { free (h); return ; }
static inline int8_t compare(treeNode *l, treeNode *r){
   return l-> count - r-> count ; // l < r if negative , l = r if 0, ...
}
// functional prototypes (functional declarations)
treeNode *newNode (uint8_t s, bool l, uint64_t c);
treeNode *delTree(treeNode *t);
void dumpTree(treeNode *t, FILE *fp);
void loadHuffmanBuffer(FILE *, uint64_t);
treeNode *loadTree (uint8_t *savedTree, uint64_t c);
void buildCode(treeNode *t, code *s, code table[256]) ;
#endif
```

This header-file pattern:

Modularize your Code

```
#ifndef MODULE_H
#define MODULE_H
  /* lots of stuff */
#endif
```

- The purpose of making it possible to mention each header file where it's needed without worrying about including it more than once.

Note: Usually .c and .h files are paired. The .h file contains the declarations necessary to use the functions in the .c file. Sometimes that's just a function declaration, and sometimes it also includes declarations of constants, structs, and other such things.

main.c

```
#include "Destroy.h"
#include <stdio.h>
#include <stdlib.h>
#include "struct.h"

int main(){
        .
        .
        printf..
        .
        .
        Destroy()
        .
        .
        return 0;
```

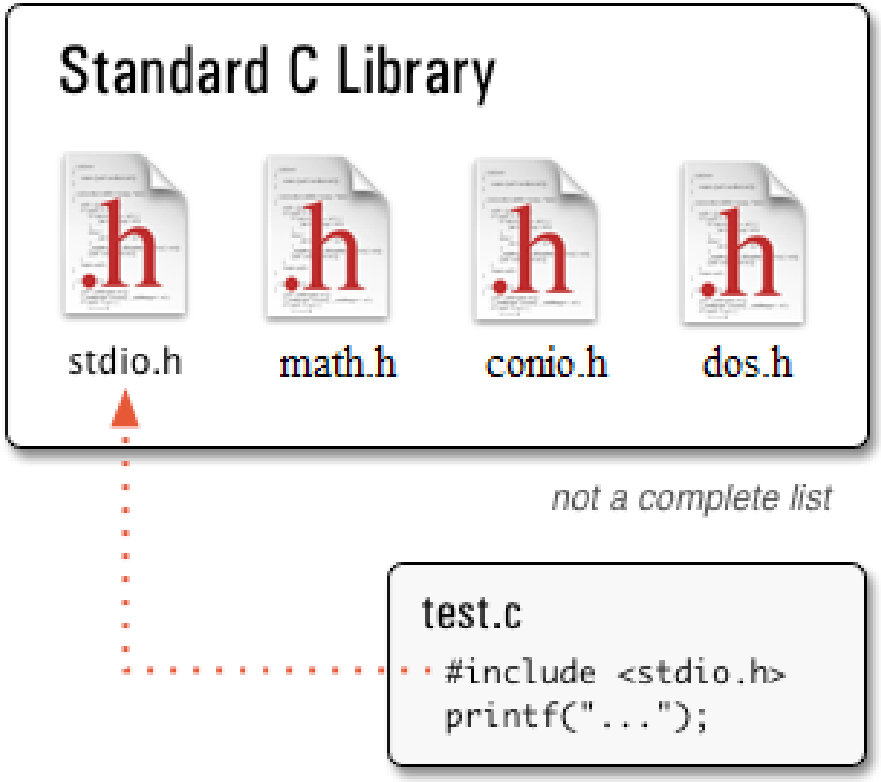
these are also used in main()

Destroy.c

```
#include "Destroy.h"
#include <stdio.h>
#include <stdlib.h>
#include "struct.h"

void Destroy(){
.
.

.
}
```

Destroy.h

```
#ifndef __Card__Destroy__
#define __Card__Destroy__


void Destroy( );


#endif /* defined(__Card__Destroy__) */
```



Header File	Description
<ctype.h>	Character testing and conversion functions
<math.h>	Mathematical functions
<stdio.h>	Standard I/O functions
<stdlib.h>	Utility functions
<string.h>	String handling functions
<time.h>	Time manipulation functions

utils.c

```
#define MODULE
#include "aa.h"


 double f1(void)
 {
   ....
 }

  int  g1(int a)
  {
   ....
  }
```

aa.h

```
#ifndef MODULE
#define EXTERN extern
#else
#define EXTERN
#endif

EXTERN double f1(void);

EXTERN int g1(int);
```

mod1.c

```
#include "aa.h"

 k = g1( i );
 yd = f1( );
```

mod2.c

```
#include "aa.h"

 j = g1(5);
```

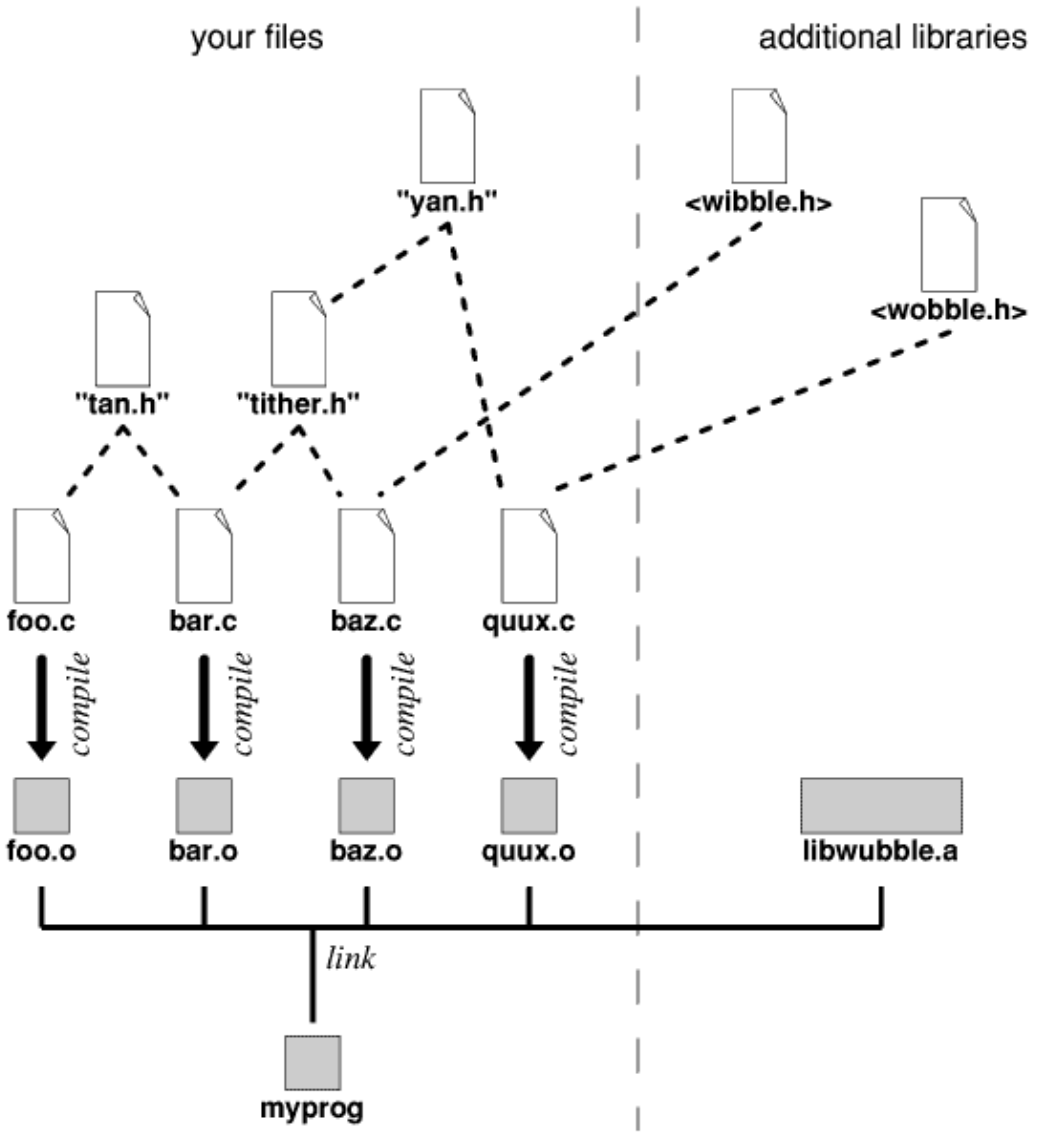
mod3.c

```
#include "aa.h"

 ydot = f1( );
```

Source dependencies

mod1.c –> (aa.h)
mod2.c –> (aa.h)
mod3.c –> (aa.h)

utils.c –> (aa.h)

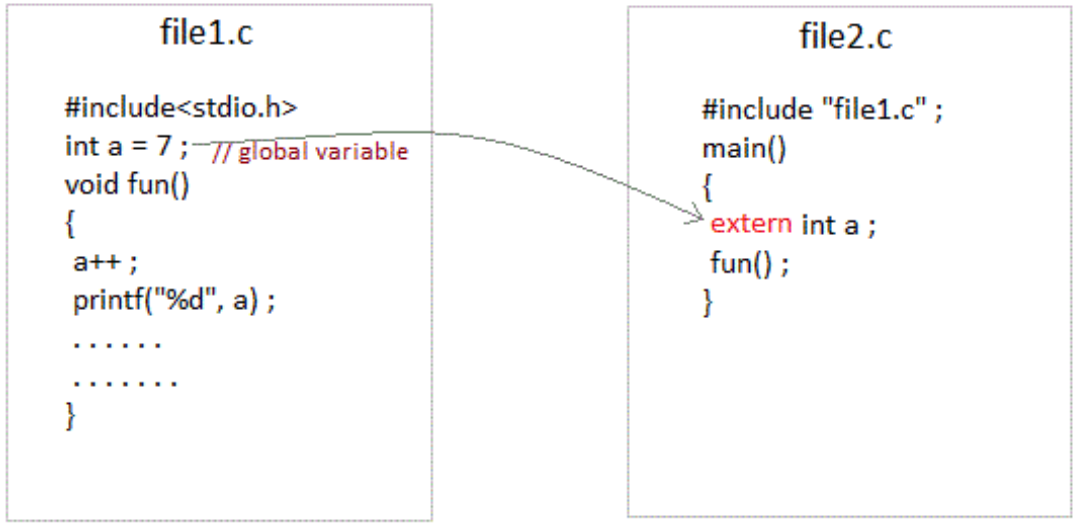

your files
additional libraries
"yan.h"
<wibble.h>
<wobble.h>
"tan.h"
"tither.h"
foo.c
bar.c
baz.c
quux.c
compile
compile
compile
compile
foo.o
bar.o
baz.o
quux.o
libwubble.a
link
myprog

LECTURE 6

External Reference

extern keyword

- The extern keyword is used to before a variable to inform the compiler that this variable(functions) is declared somewhere else.
- The extern declaration does not allocate storage for variables (or functional definition).



global variable from one file can be used in other using **extern** keyword.

Demo Program: a.h+a.c+

Go gcc!!!

build.bat

```
gcc a.c b.c -o b
```

b.c

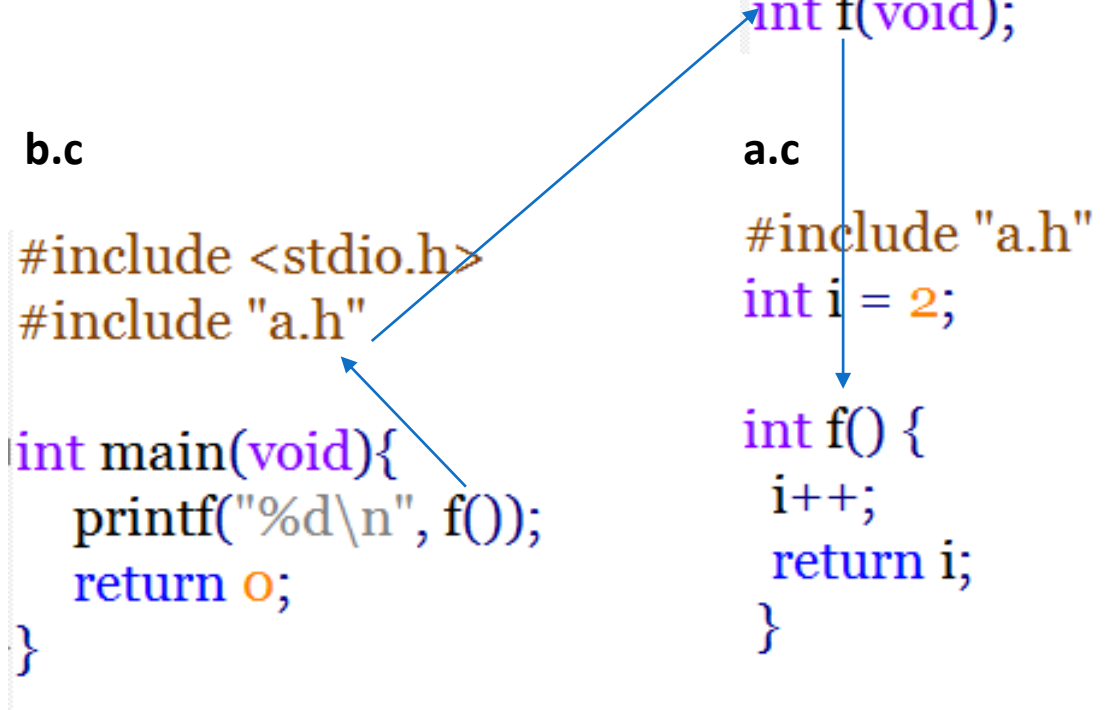
```c
#include <stdio.h>
#include "a.h"

int main(void){
    printf("%d\n", f());
    return 0;
}
```

a.h

```c
extern int i;
int f(void);
```

a.c

```c
#include "a.h"
int i = 2;

int f() {
  i++;
  return i;
 }
```



extern keyword

- The extern in the header is useful, because it tells the compiler during the link phase, "this is a declaration, and not a definition". If I remove the line in a.c which defines i, allocates space for it and assigns a value to it, the program should fail to compile with an undefined reference. This tells the developer that he has referred to a variable, but hasn't yet defined it. If on the other hand, I omit the "extern" keyword, and remove the int i = 4 line, the program still compiles - i will be defined with a default value of 0.
- File scope variables are implicitly defined with a default value of 0 or NULL if you do not explicitly assign a value to them - unlike block-scope variables that you declare at the top of a function. The extern keyword avoids this implicit definition, and thus helps avoid mistakes.
- For functions, in function declarations, **the keyword is indeed redundant**. Function declarations do not have an implicit definition.

Example 1:

```
int var;
int main(void)
{
   var = 10;
   return 0;
}
```

Analysis: This program is compiled successfully. Here var is defined (and declared implicitly) globally.

Example 2:

```
extern int var;
int main(void)
{
  return 0;
}
```

Analysis: This program is compiled successfully. Here var is declared only. Notice var is never used so no problems.

Example 3:

```
extern int var;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: This program throws error in compilation. Because var is declared but not defined anywhere. Essentially, the var isn't allocated any memory. And the program is trying to change the value to 10 of a variable that doesn't exist at all.

Example 4:

```
#include "somefile.h"
extern int var;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: Supposing that somefile.h has the definition of var. This program will be compiled successfully.

Example 5:

extern actually means “defined somewhere”.

```
extern int var = 0;
int main(void)
{
 var = 10;
 return 0;
}
```

Analysis: Guess this program will work? Well, here comes another surprise from C standards. They say that..if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined. Therefore, as per the C standard, this program will compile successfully and work.

extern keyword

- The extern keyword takes on different forms depending on the **environment (OS+Compiler)**.
- If a declaration is available, the extern keyword takes the linkage as that specified earlier in the translation unit.
- In the absence of any such declaration, extern specifies external linkage.

```
static int g();
extern int g(); /* g has internal linkage */

extern int j(); /* j has tentative external linkage */

extern int h();
static int h(); /* error */
```

Defined somewhere already

extern Keyword

1. Declaration can be done any number of times but definition only once.

2. “extern” keyword is used to extend the visibility of variables/functions().

3. Since functions are visible through out the program by default. The use of extern is not needed in function declaration/definition. Its use is **redundant**. (It works more like comment, but I like it.)

4. When extern is used with a variable, it’s only declared not defined.

5. As an exception, when an extern variable is declared with initialization, it is taken as definition of the variable as well.

LECTURE 7

Scope of Variables and Functions

Terminologies

Binding of a variable to a memory location: The association of a variable to a memory location is called binding.

Scope of a variable: The region or life time that the binding is in power is called the scope of a variable.

Life of a variable: The life of a variable may be longer than the scope of a variable, because some variable may be put into “inactive” while some other variable of the same name are in power. Once that other variable’s binding finished. This variable may be brought into power again. The life time of a variable will be from the creation of a variable until the dis-allocation of the memory assignment.

C Scope Rules

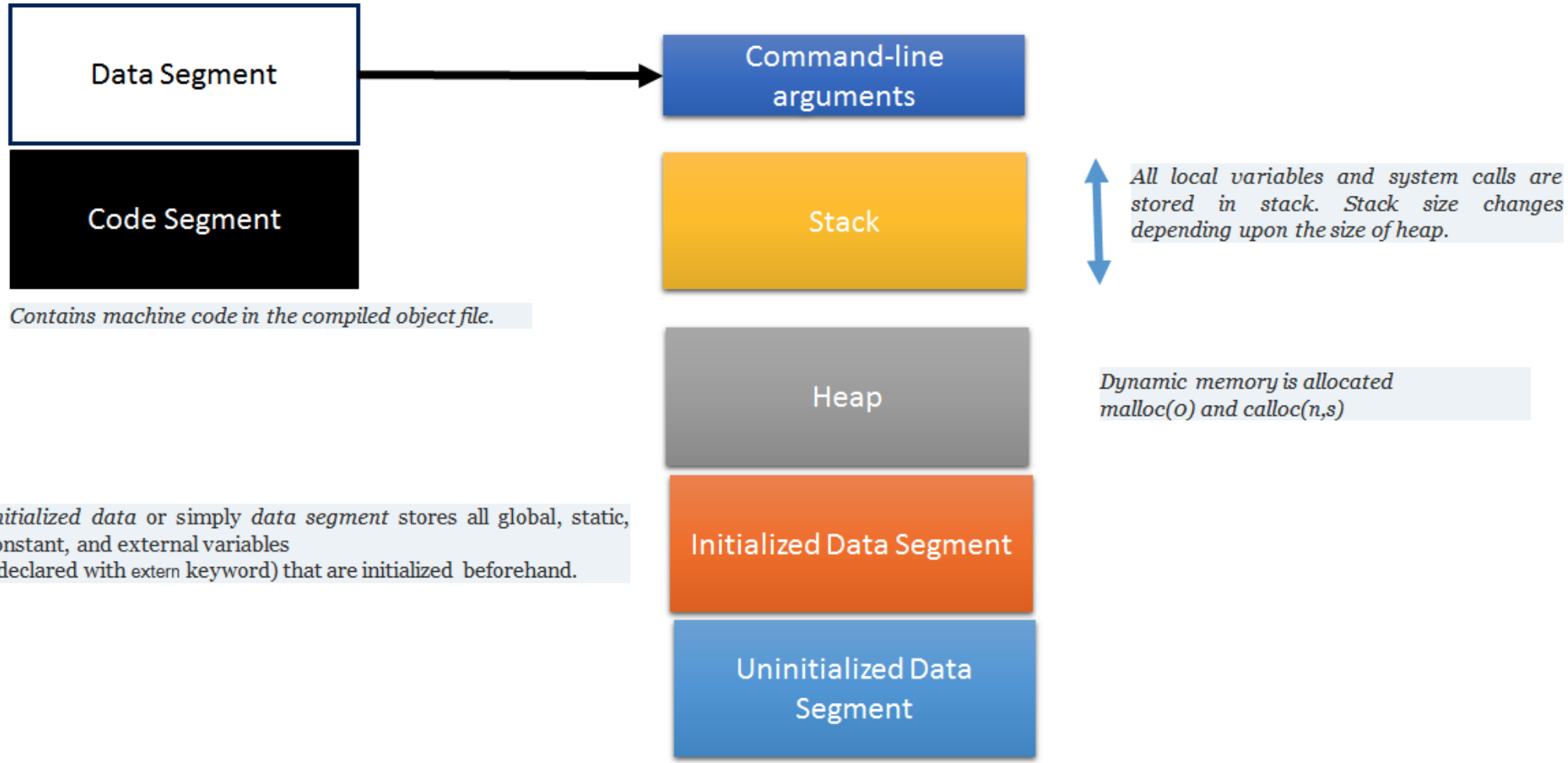
A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables and **formal** parameters.

C memory Model

Data segment stores program data. It is again divided into

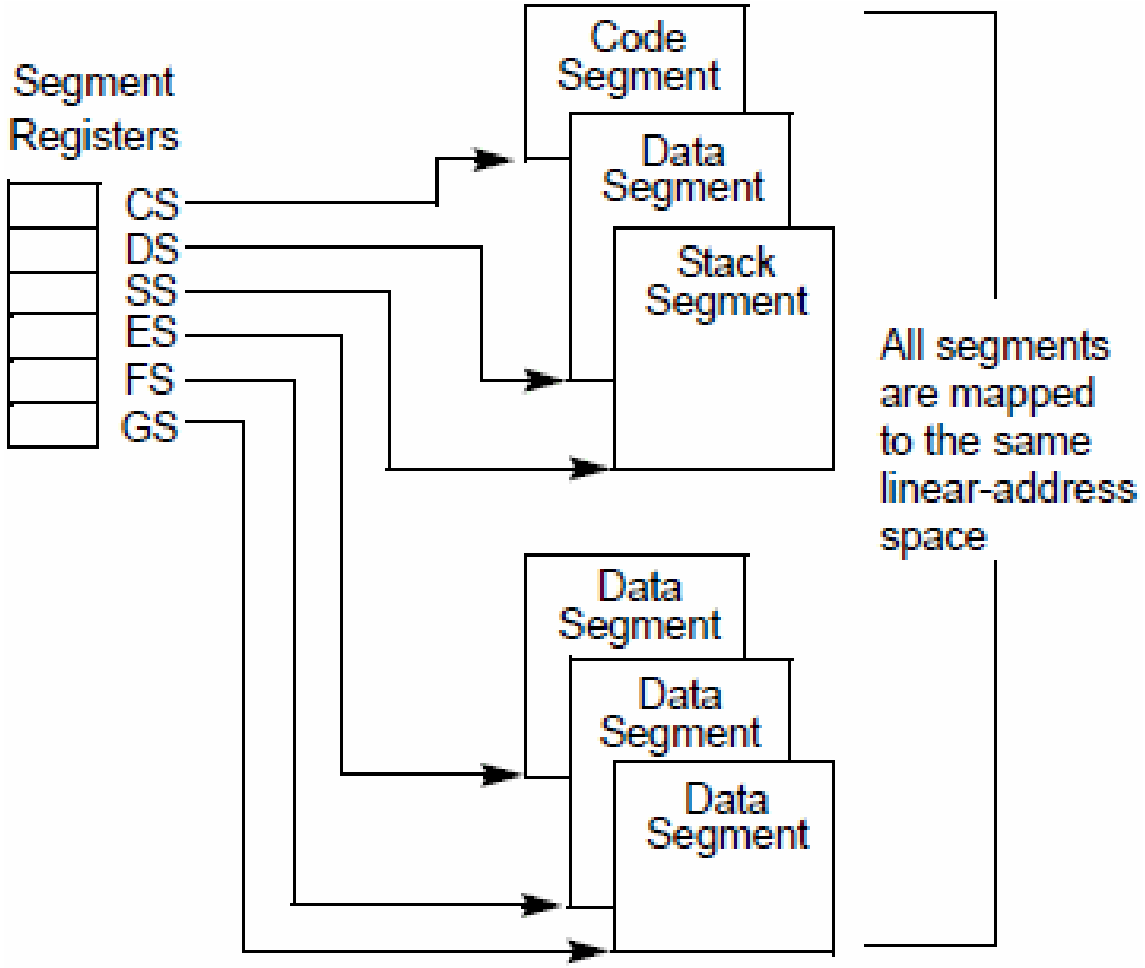


Contains machine code in the compiled object file.

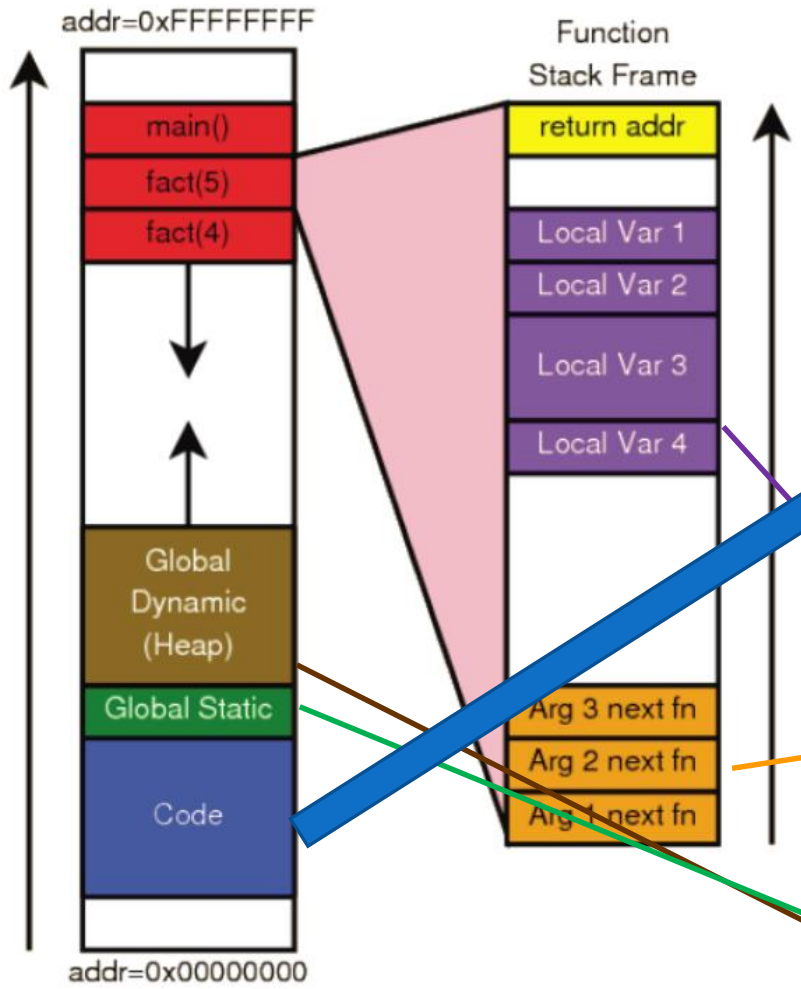
All local variables and system calls are stored in stack. Stack size changes depending upon the size of heap.

Dynamic memory is allocated malloc(0) and calloc(n,s)

Initialized data or simply *data segment* stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand.



Use of Segment Registers in Segmented Memory Model



C Language Memory Model

style1.c

```c
#include <stdio.h>
void add_print(int , int); //function declaration
int main(){
   int a=4;
   int b=5;
   printf("Entering 'add_print' function\n");
   add_print(a,b);
   printf("Just came from 'add_print' function\n");
return 0;
}
//function definition
void add_print(int val1,int val2){
   int c;
   printf("The two values entered are:%d,%d \n",val1,val2);
   c=val1+val2;
   printf("Sum of numbers entered is:%d \n",c);
}
```

```c
static int a;
int *ptr = calloc(1, sizeof(int));
```

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code.

Local variables are not known to functions outside their own.

The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

Demo Program: local.c

```c
#include <stdio.h>

int main () {
  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
  return 0;
}
```

local.c

Formal Parameters

Formal parameters, are treated as **local variables** with-in a function and they take precedence over global variables.

Demo Program: formal.c

Go gcc!!!

```c
#include <stdio.h>
/* global variable declaration */
int a = 20;
int main () {
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;
  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);
  return 0;
}

/* function to add two integers */
int sum(int a, int b) {
   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);
   return a + b;
}
```

Global Variables

- Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
- A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.
- A program can have same name for local and global variables but the value of local variable inside a function will take preference.

Demo Program:

global1.c and global2.c

Go gcc!!!

LECTURE 8

Static Variable and Functions

Static Variables and Functions

In C language, the life time and scope of a variable is defined by its storage class.

The following are four types of storage class available in C language.

- auto
- register
- extern
- static

In this lecture, we will discuss the ‘static’ storage class and explain how to use static variables and static functions in C with some sample code snippets.

Impact on Life Time

- static variables are those variables whose life time remains equal to the life time of the program. Any local or global variable can be made static depending upon what the logic expects out of that variable.
- Demo Program: staticx.c

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\static>staticx
f=3
Main=3
f=4
```

```c
#include<stdio.h>
static x=3;

void func(){
   printf("f=%d\n", x);
}
int main(void){
   func();
   printf("Main=%d\n", x);
   x=4;
   func();
   return 0;
}
```

Impact on Scope

- In case where code is spread over multiple files, the static storage type can be used to limit the scope of a variable to a particular file.
- For example, if we have a variable ‘count’ in one file and we want to have another variable with same name in some other file, then in that case one of the variable has to be made static.

Demo Program: staticy.c

```c
#include<stdio.h>
static x=3;
int y=3;
void func(){
    int y=10;
    printf("f(x)=%d\n", x);
    printf("f(y)=%d\n", y);
}
int main(void){
    func();
    printf("Main(x)=%d\n", x);
    printf("Main(y)=%d\n", y);
    x=4; y=4;
    func();
    return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\static>staticy
f(x)=3
f(y)=10
Main(x)=3
Main(y)=3
f(x)=4
f(y)=10
```

Global static and local static

- A static variable inside a function keeps its value between invocations.
- A static global variable or a function is "seen" only in the file it's declared in.

- Demo Program: staticf.c

```c
#include <stdio.h>
static int callNo =0;
void f(){
   static x=0;
   callNo++;
   x++;
   printf("x=%d\n", x);
}
void g(){
   static y=0;
   callNo++;
   y++;
   printf("y=%d\n", y);
}

int main(void){
  f(); g(); f(); g(); f(); g(); printf("Total Function Calls=%d\n", callNo);
  return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\static>staticf
x=1
y=1
x=2
y=2
x=3
y=3
Total Function Calls=6
```

Static Functions

By default any function that is defined in a C file is **extern**. (Defined somewhere, considered as **public**) This means that the function can be used in any other source file of the same code/project (which gets compiled as separate translational unit).

Now, if there is a situation where the access to a function is to be limited to the file in which it is defined or if a function with same name is desired in some other file of the same code/project then the functions in C can be made static. (Defined in the module, considered as **private**)

Demo Program:

main.c+a.h+a.c+b.h+b.c

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\static>main
Main()================
main's x=3
main's y=7
main's z=5
A's other() ===========
other's static x=4
other's           y=7
other's           z=6
A's f()
Stranger() ================
Stranger's x=15
Stranger's y=8
Stranger's z=7
B's g()
other calling
B's g()
```

```c
#include <stdio.h>
#include "a.h"

static int x = 3;
int z = 5;
static void f(){
    printf("Main's static f() \n");
}

int main(){
    printf("Main()================\n");
    printf("main's x=%d\n", x);
    y++;
    printf("main's y=%d\n", y);
    printf("main's z=%d\n", z);
    other();
    return 0;
}
```

```c
#ifndef _A_
#define _A_
extern int y;
extern void other(void);
#endif
```

```c
#include <stdio.h>
#include "a.h"
static x = 4;
int y=6;
extern int z;
static f(){
    printf("A's f() \n");
}
void other(){
    printf("A's other() =========== \n");
    printf("other's static x=%d\n", x);
    printf("other's          y=%d\n", y);
    z++;
    printf("other's          z=%d\n", z);
    f();
    stranger();
    printf("other calling\n");
    g();
}
```

```c
#include <stdio.h>
#include "b.h"
#include "a.h"
int x = 15;
//int z=100;
extern int z;
void g(){
    printf("B's g()\n");
}
void stranger(){
    printf("Stranger() ================\n");
    printf("Stranger's x=%d\n", x);
    y++;
    printf("Stranger's y=%d\n", y);
    z++;
    printf("Stranger's z=%d\n", z);
    g();
}
```

```c
#ifndef _B_
#define _B_
#endif
```


All functions are assigned to extern by default.
Variables are module-based (non-external)

Summary for static and extern (I)

1. static int x in main.c and a.c are different variables. (Their scopes are valid their own module file).

2. int y defined in a.c. It is shared by b.c and main.c. It is made available for other module by the extern declaration in a.h.

3. int z defined in main.c It is shared with a.c and b.c. In these files, they have their own extern int z declaration. There is not .h file for these shared variable definition. It is also OK, but harder to maintain the program.

Summary for static and extern (II)

4. static f()'s have different definitions in main.c and a.c.

5. g() is defined in b.c and it is shared in a.c even without the keyword extern.

6. The variable/function in the modules somewhat like data abstraction.

Static works like private (in OOP). Extern works like public (in OOP). But not 100% of the same meaning. static to function is an exception.

Keyword	Variables	Functions
extern	Visible to the included modules	Visible to all modules (extern)
(default)	Visible to the module (static)	Visible to all modules (extern)
static	Visible to the module (static)	Visible to the module (static)

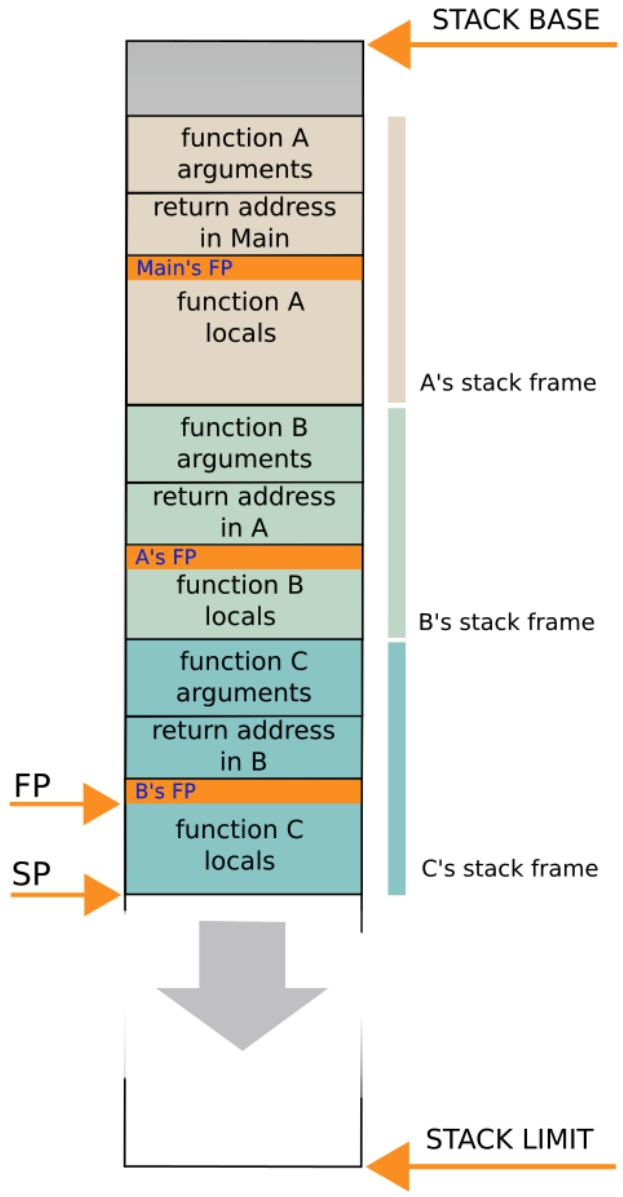
LECTURE 9

Parameters and Return Value (Non-void and Non-int Functions)

Pass or Return by Primitive Data Type

Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFF ($4.45015E-308_{10}$)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal



Arrays as function parameters

- When passing arrays to functions (as parameters) things can get a little confusing. It is not possible to pass an array by-value to a function, so the function process_array() below does not make a copy of the array:

```
void process_array(uint32_t array[10])
{
  for (int i = 0; i < 10; ++i)
  {
    printf("%d", array[i]);
  }
}

int main(void)
{
  uint32_t a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  process_array(a);
  return 0;
}
```

- The array parameter degrades to a pointer – the address of the first element; so we could (and many C programmers do) just as legitimately write the following and get the same result:

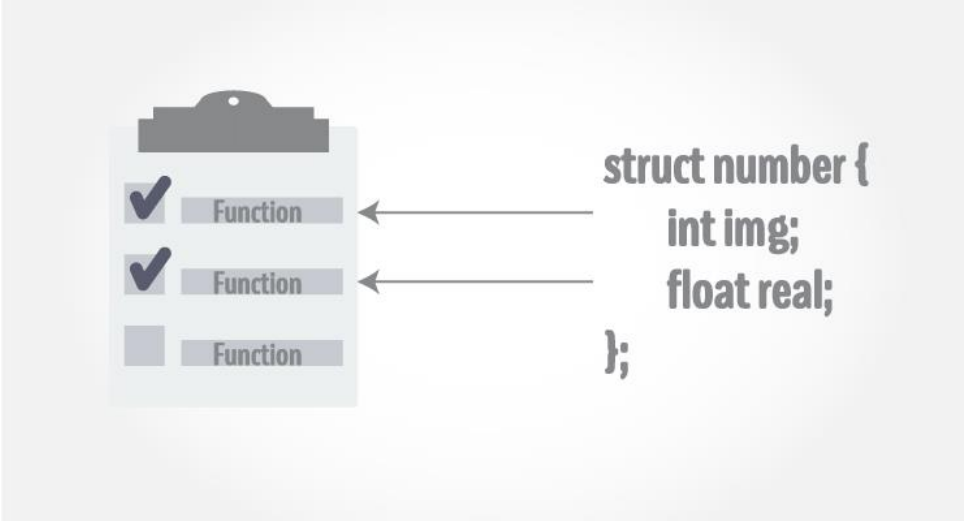
```
void process_array(uint32_t *array)
{
  for (int i = 0; i < 10; ++i)
  {
    printf("%d", array[i]);
  }
}
```

Arrays as function parameters

In fact, all these declarations for process_array() are semantically identical; the code generated is the same in each case:

```
void process_array(uint32_t array[5]);  // The 5 is ignored!
void process_array(uint32_t array[]);
void process_array(uint32_t *array);
```

Passing Structure to a Function



In C, structure can be passed to functions by two methods:

1. Passing by value (passing actual value as argument)
2. Passing by reference (passing address of an argument)

Passing Structure by Value

- A structure variable can be passed to the function as an argument as a normal variable.
- If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.
- C program to create a structure student, containing name and roll and display the information.

Pass by Value is Copying Values of the Struct Value

- Two value of the same structure type can be copied in the same way as ordinary variables.
- If student1 (a struct) and students belong to the same structure, then the following statements are valid:
  - student1 = student2;
  - student2 = student1;
- However, the equality check by == operator such as:
  - student1 == student2
  - student1 != student2
- Are not permitted. (In Java, the equality for object need equals() function).
- If we need to compare the structure variables, we may do so by comparing members individually.

Demo Program:

passValue.c

Go gcc!!!

Passing structure by reference

- The memory address of a structure variable is passed to function while passing it by reference.
- If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.
- C program to add two distances (feet-inch system) and display the result without the return statement.

Demo Program:

passReference.c

Go gcc!!!

LECTURE 10

Auto/Register Storage Classes and Code Block

Code Block

- A code block in C language is defined by two parentheses.

```
{  /*Put Your Code Here*/
}
```

- A code block forms a separate local variable domain.
- A code block includes multiple statements (0 to many).

Storage classes

Specifiers	Lifetime	Scope	Default initializer
`auto`	Block (stack)	Block	Uninitialized
`register`	Block (stack or CPU register)	Block	Uninitialized
`static`	Program	Block or compilation unit	Zero
`extern`	Program	Block or compilation unit	Zero
(none)[1]	Dynamic (heap)		Uninitialized

[1] Allocated and deallocated using the `malloc()` and `free()` library functions.

Note: Every object has a storage class. This specifies most basically the storage duration, which may be static (default for global), automatic (default for local), or dynamic (allocated), together with other features (linkage and register hint).

auto Storage Class Specifier

- Variables declared within a block by default have automatic storage, as do those explicitly declared with the **auto** or register storage class specifiers.
- The auto and register specifiers may only be used within functions and function argument declarations; as such, the auto specifier is always redundant.

Demo Program:

auto.c

Go gcc!!!

```c
#include <stdio.h>

int main(){
    auto int number = 5;
    {  // a code block set up a auto variable domain.
       auto int number = 20;
       printf("inner number: %d", number);
    }
    printf("\n");
    printf("outer number: %d", number);
    return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\auto>auto
inner number: 20
outer number: 5
```

register storage class specifier

- The central processor can hold a very small amount of data. This data is stored in places called registers. Data saved there, is accessed faster than data stored in memory. For that reason programs can load their most-frequently accessed data in the CPU registers.
- This is the very idea of the register keyword in C. It tells the compiler that we will work frequently with the current variable and it should be stored for fast access. Note that this is only a “hint” to the compiler and it could ignore it.

```c
#include<stdio.h>


int main()
{
 int a = 5;

 register int b = 20;
 printf("the value of b = %d\n",b);

return 0;
}
~
~
~
~
~
~
~
```

Restrictions

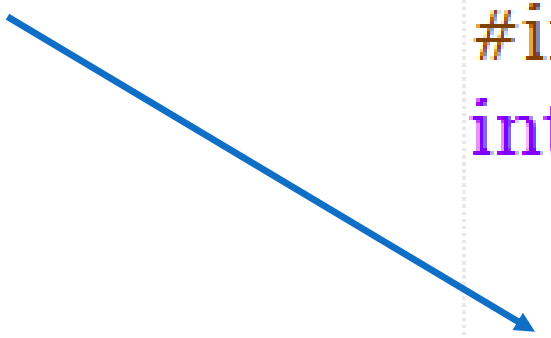
- Register is a storage class specifier. We cannot use more than one storage class specifier for one variable. This means that the following declarations are not correct:

```
register static int number;

register auto int number;

register extern int number;
```

- Placing a variable in a processor register means that it does not have a memory address, because it is not in the memory. Therefore, in C, if you declare a variable with the register keyword, you **cannot** access its address with the & operator.
- Even if the variable is placed in memory, the compiler should not allow you to access its address. You can use this to make sure that some variable will not be changed outside of its function.

Erroneous example

- There is no address for register variables.

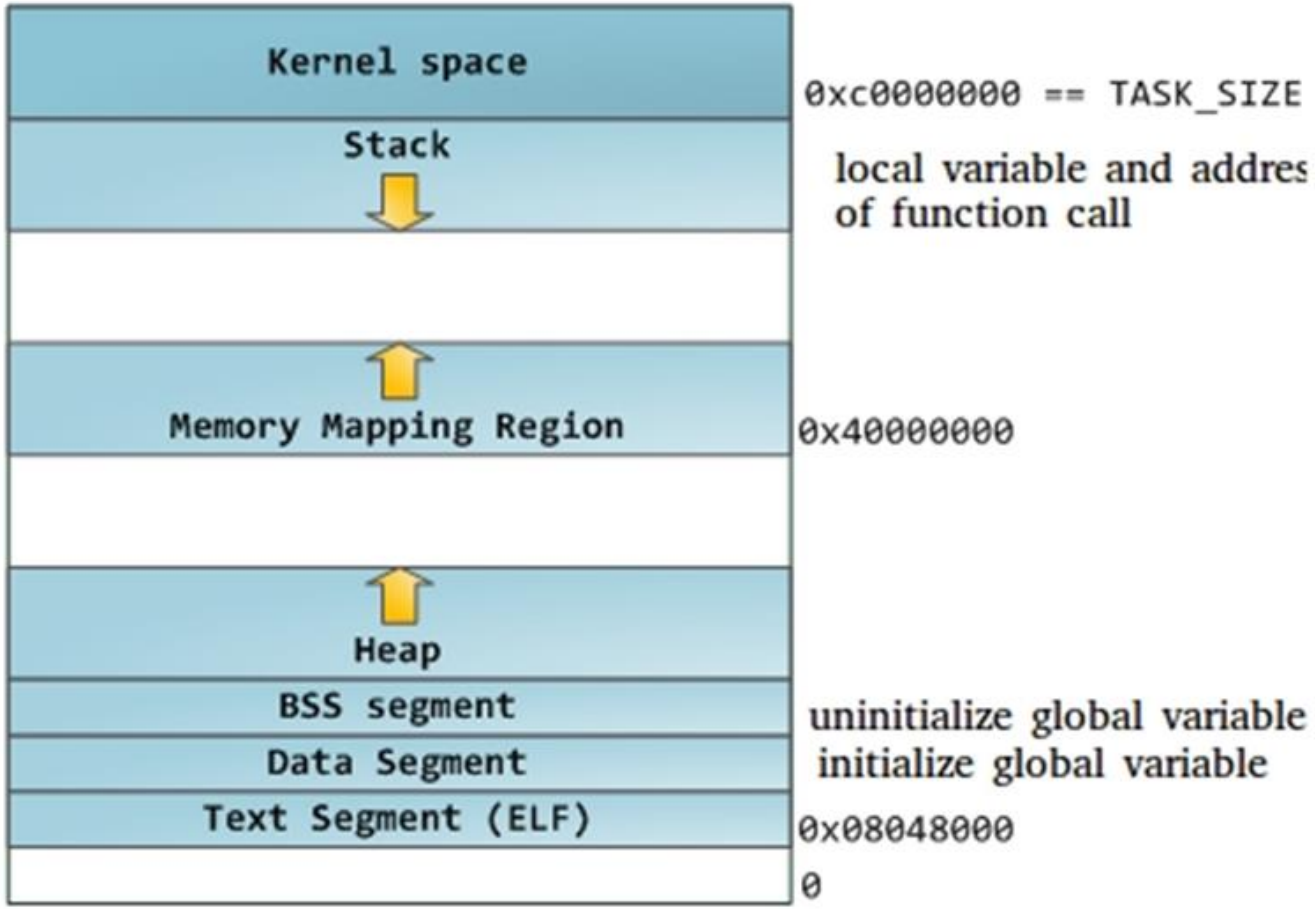


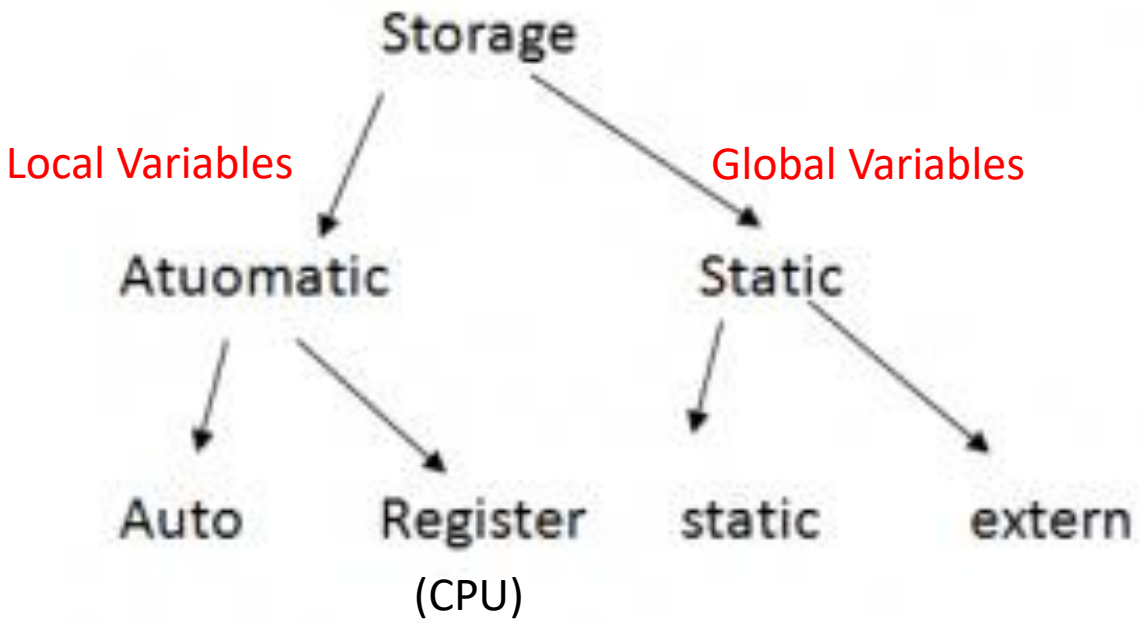
```c
#include <stdio.h>
#include <stdlib.h>
int main(){
    register int number = 1;
    scanf("%d", &number);
    printf("%d", number);
    return 0;
}
```

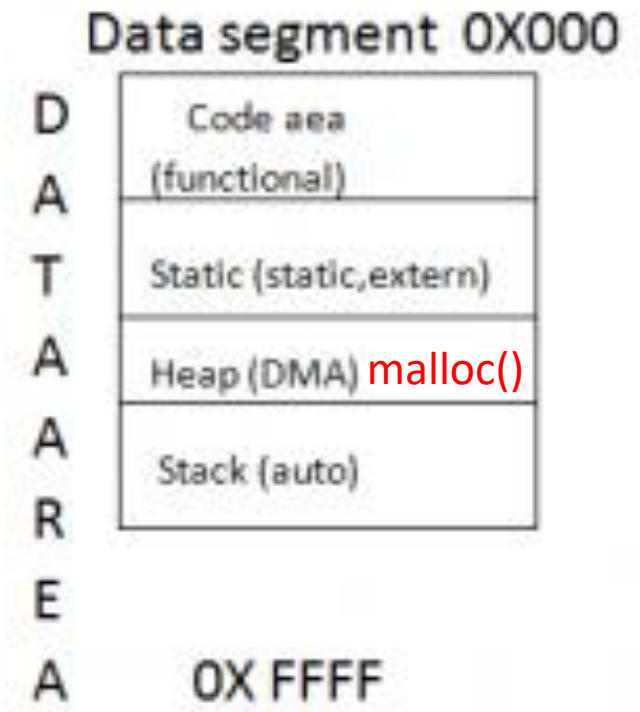
LECTURE 11

Initialization of Local Variables

Memory mapping in c




Storage
Local Variables
Global Variables
Atuomatic
Static
Auto
Register
static
extern
(CPU)


Data segment 0X000
DATA AREA
Code aea (functional)
Static (static,extern)
Heap (DMA) malloc()
Stack (auto)
0X FFFF

Serial No	Storage Class	Storage Place	Initial/Default Value	Scope	Lifetime
1.	extern	Data Segment	Zero	Global	Till the end of the main program. Variable definition might be anywhere in the C program
2.	static	Data Segment	Zero	Local	Retains the value of the variable between different function calls.
3.	auto	Stack memory	Garbage Value	Local	Within the function only.
4.	register	Stack memory or CPU Register	Garbage Value	Local	Within the function only.

Integer Literals

- An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, 0b or 0B for binary and nothing for decimal.
- An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

```
85          /* decimal */
0213        /* octal */
0x4b        /* hexadecimal */
30          /* int */
30u         /* unsigned int */
30l         /* long */
30ul        /* unsigned long */
```

Floating-point Literals

- A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.
- While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character Constants

- Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of char type.
- A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

String Literals

- String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.
- You can break a long line into multiple lines using string literals and separating them using white spaces.
- Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"


"hello, \

dear"


"hello, " "d" "ear"
```

Initialization of Array Elements

A. Method 1 : Array Size Specified Directly

```
int num[5] = {2,8,7,6,0};
```

In the above example we have specified the size of array as 5 directly in the initialization statement.Compiler will assign the set of values to particular element of the array.

```
num[0] = 2
num[1] = 8
num[2] = 7
num[3] = 6
num[4] = 0
```

B. Method 2 : Size Specified Indirectly

In this scheme of compile time Initialization, We does not provide size to an array but instead we provide set of values to the array.

```
int num[] = {2,8,7,6,0};
```

Declaring a Structure Type

```
struct student
{
        int roll_no;
        char name[30];
        float percentage;
};
```

Declaring a Structure Variable

```
struct student s1,s2,s3;
        (or)
struct student
{
        int roll_no;
        char name[30];
        float percentage;
}s1,s2,s3;
```

Initialization of structure

Initialization of structure variable while declaration :

```
struct  student  s2 =  {  1001, " K.Avinash ",
                  87.25 } ;
```

Initialization of structure members individually :

```
s1. roll_no = 1111;
strcpy ( s1. name , " B. Kishore " ) ;
s1.percentage = 78.5 ;
```

membership operator

Reading values to members at runtime:

```
struct student s3;
printf("\nEnter the roll no");
scanf("%d",&s3.roll_no);
printf("\nEnter the name");
scanf("%s",s3.name);
printf("\nEnter the percentage");
scanf("%f",&s3.percentage);
```

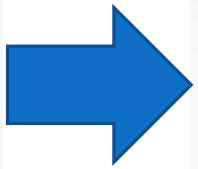
LECTURE 12

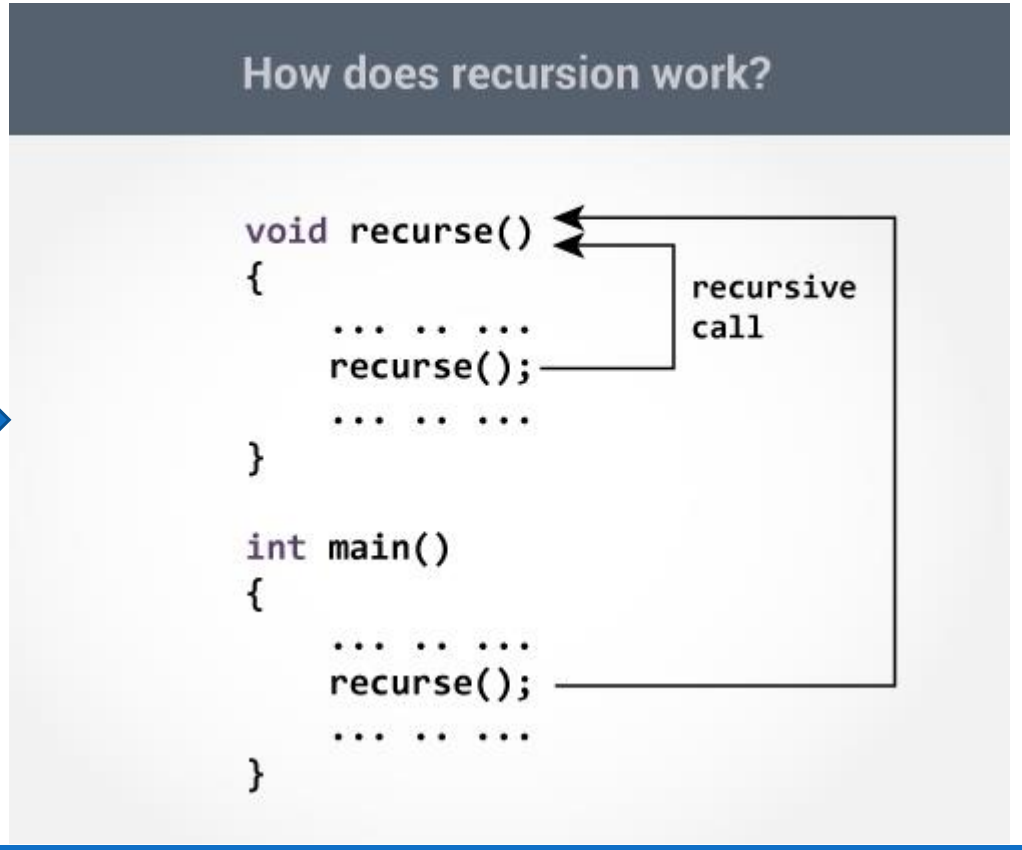
Introduction to Recursion

C Programming Recursion

A function that calls itself is known as a recursive function. And, this technique is known as **recursion**.

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```





Recursion

- Recursion is when a function calls itself.
  - Great Utility
  - Makes the code easier
- Requirements to use recursion
  - A condition to **cease** at
    - otherwise the program would never terminate
    - the condition is usually written at the beginning of the recursive method

Recursion and Iteration

Example:

```
/* non-recursive */
int fact(int n) {
      int t, answer;
      answer = 1;
      for(t=1; t<=n; t++)
      answer=answer*(t);
      return(answer);
}
```

```
/* recursive */
int factr(int n) {
      int answer;
      if(n==1) return(1);
      answer = factr(n-1)*n; /* recursive call */
      return(answer);
}
```

Design of Recursive Program

Sum(n): sum of 1 to the number n.

1. Finding the recursive formula and stop condition

   Recursive Condition: $Sum(n) = \sum_{k=1}^{n} k = n + \sum_{k=1}^{n-1} k$

   Stop Condition: $Sum(1) = 1$

2. Formulate it into recursive C code

```
int sum(int n) { if (n==1) return 1;  return n+sum(n-1); }
```

Demo Program: sumN.c

Go gcc!!!

```c
#include <stdio.h>

int sum(int n){
  if (n==1) return 1;
  return n+sum(n-1);
}

int main(void){
  printf("Sum from 1 to %d is %d\n", 10, sum(10));
  return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\sumN>sumN
Sum from 1 to 10 is 55
```

Tail Recursion

- In computer science, *a tail call is a subroutine call performed as the **final action** of a procedure*.
- If a tail call might lead to the same subroutine being called again later in the call chain, the subroutine is said to be tail-recursive, which is a special case of recursion.
- Tail recursion (or tail-end recursion) is particularly useful, and often easy to handle in implementations.

- A tail recursive function call allows the compiler to perform a ***special optimization*** which it normally can not with regular recursion.
- Advantage of using tail-recursion so that the compiler optimize the code and convert it to a **non-recursive** code.
- Advantage of non-recursive code over recursive one, the non-recursive code requires less memory to execute than a recursive one. This is because of idle stack frames that the recursion consumes.

Demo Program:

sumTail.c

Go gcc!!!

```c
#include <stdio.h>
int sumt(int n, int s){
  if (n==1) return s+1;
  int ss = s +n;
  return sumt(n-1, ss);
}
int sum(int n){
    return sumt(n, 0);
}
int main(void){
  printf("Sum from 1 to %d is %d\n", 10, sum(10));
  return 0;
}
```

```
C:\Eric_Chou\C Course\C Programming Essentials\CDev\Ch6\sumN>sumTail
Sum from 1 to 10 is 55
```

Higher Degree Recursion

Because of the recursive nature, any recursive program of higher than first degree will be intractable. $O(n^2)$ or higher.

Therefore, usually recursive program of higher degree is used for prototyping purpose. For real-world computation, it is usually recommended to convert the program to iterative counterparts.

Demo Program:

Fibonacci number generator.

fib(n)

In fib.c and/or fibt.c(pseudo-tail recursive format, co-routine.)

```c
#include <stdio.h>
int fib(int n){
  if (n== 0 || n == 1) return 1;
  return fib(n-1)+fib(n-2);
}
int main(void){
  printf("Fibonacii(5)=%d\n", fib(5));
  return 0;
}
```

fib.c

```c
#include <stdio.h>
int fibt(int n, int s1, int s2){
  if (n==0 || n==1) return 1;
  int sum=s1+s2;
  return sum;
}

int fib(int n){
    if (n==0 || n==1) return 1;
    return fibt(n, fib(n-1), fib(n-2));
}
int main(void){
  int i;
  printf("Enter an integer: ");
  scanf("%d", &i);
  printf("Fibonacii(%d)=%d\n", i, fib(i));
  return 0;
}
```

fibt.c

LECTURE 13

Inclusion (#include)

C Language: #include Directive

This C tutorial explains how to use the #include preprocessor directive in the C language.

Description

In the C Programming Language, the #include directive tells the preprocessor to insert the contents of another file into the source code at the point where the #include directive is found. Include directives are typically used to include the C header files for C functions that are held outside of the current source file.

Syntax

The syntax for the #include directive in the C language is:

#include <*header_file*>

OR

#include "*header_file*"

header_file

The name of the header file that you wish to include. A header file is a C file that typically ends in ".h" and contains declarations and macro definitions which can be shared between several source files.

Note

•The difference between these two syntaxes is subtle but important. If a header file is included within <>, the preprocessor will search a predetermined directory path to locate the header file. If the header file is enclosed in "", the preprocessor will look for the header file in the same directory as the source file.

#include “filename.h”

Search project directory

- The ‘#include’ directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file.
- The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the ‘#include’ directive.

Preprocessor and Header Expansion

For example, if you have a header file **header.h** as follows,

```
char *test (void);
```

and a main program called **program.c** that uses the header file, like this,

```
int x;
#include "header.h"

int main (void){
  puts (test ());
}
```

the compiler will see the same token stream as it would if **program.c** read

```
int x;
char *test (void);
Int main (void){
  puts (test ());
}
```

#include <stdio.h>

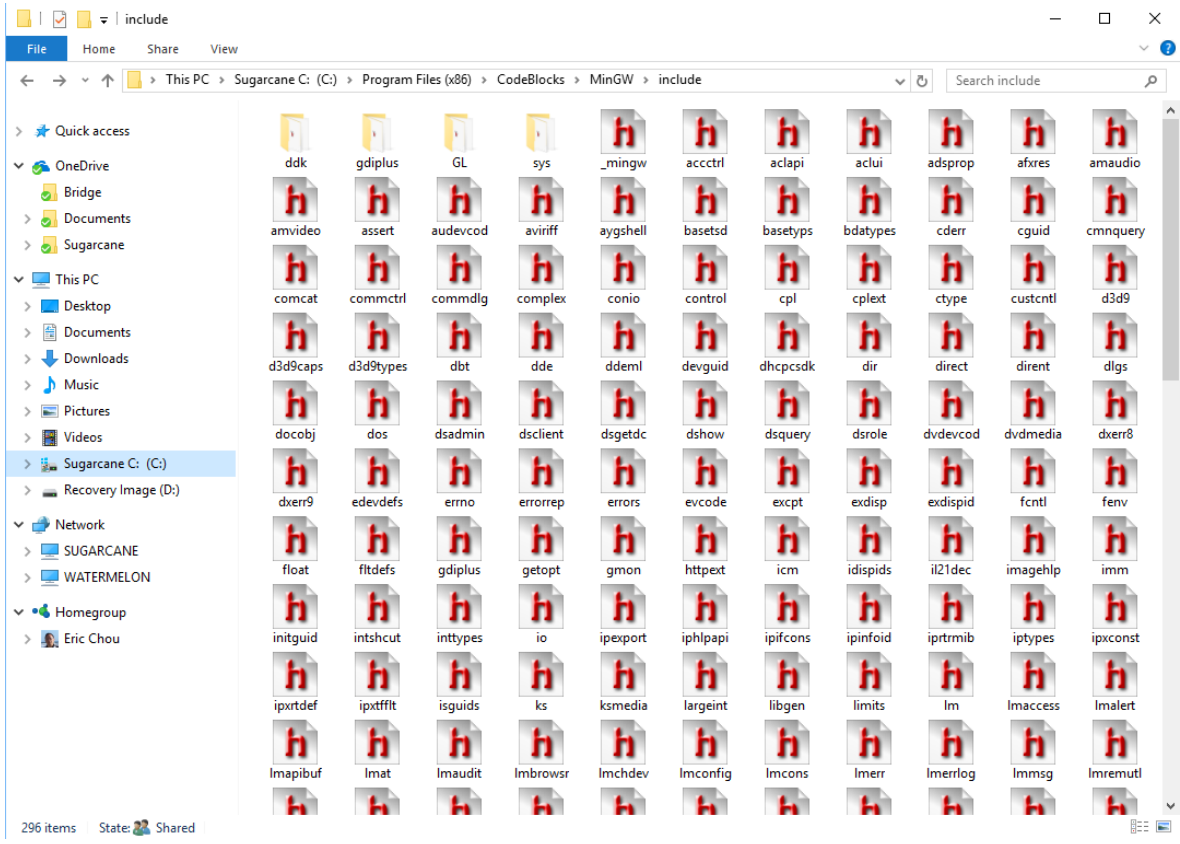
Standard Inclusion

Standard Include Files:

(Windows) C:\Program Files (x86)\CodeBlocks\MinGW\include

(Unix) /usr/include

- For the angle-bracket form #include <file>, the preprocessor's default behavior is to look only in the standard system directories.
- The most commonly-used option is **-Idir**, which causes **dir** to be searched after the current directory and ahead of the standard system directories.
- You can specify multiple **-I** options on the command line, in which case the directories are searched in left-to-right order.



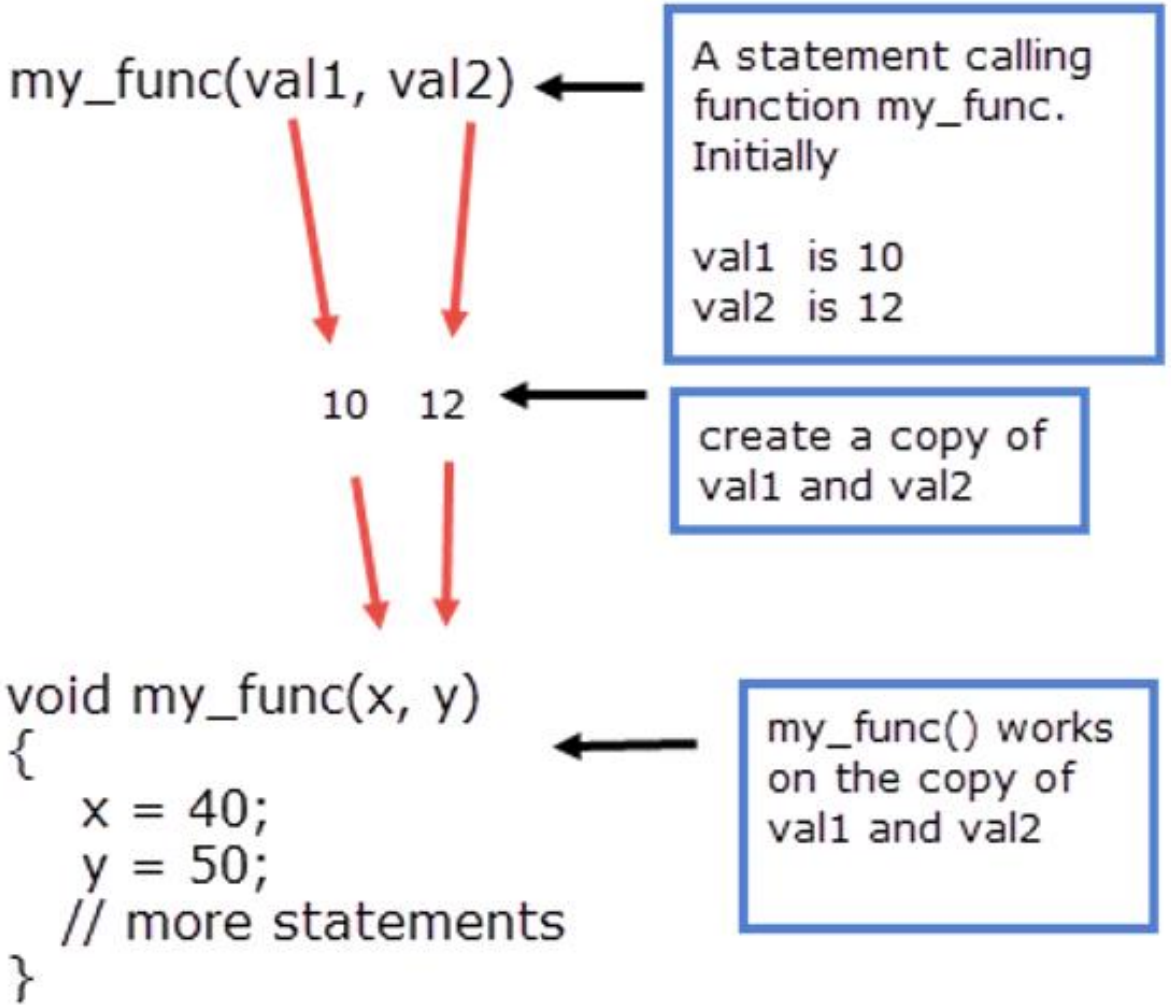
Header File	Type of Functions
<assert.h>	Diagnostics Functions
<ctype.h>	Character Handling Functions
<locale.h>	Localization Functions
<math.h>	Mathematics Functions
<setjmp.h>	Nonlocal Jump Functions
<signal.h>	Signal Handling Functions
<stdarg.h>	Variable Argument List Functions
<stdio.h>	Input/Output Functions
<stdlib.h>	General Utility Functions
<string.h>	String Functions
<time.h>	Date and Time Functions

Header Files that are Frequently Used

LECTURE 14

Call by value (copy of value)

C Supports Only Call by Value Scheme



Note:
Call by reference in C is in a way of passing pointer (address) value.

Call by value vs call by reference

Call by value	Call by reference
It consumes more memory space because formal parameter also occupies memory space.	It consumes less memory space because irrespective of data type of the actual arguments, each pointer occupies only 4 bytes
It takes more time to execute because the values are copied.	It takes less time because no values are copied.

Call by Value (Copy of Value from caller to callee)

Transfer of value from one variable to the other.

Demo Program:

callByValue.c

```c
int main(){
    int x = 10, y = 20;
    printf("Initial value of x = %d\n", x);
    printf("Initial value of y = %d\n", y);
    printf("\nCalling the function\n");
    try_to_change(x, y);
    printf("\nValues after function call\n\n");
    printf("Final value of x = %d\n", x);
    printf("Final value of y = %d\n", y);
    // signal to operating system program ran fine
    return 0;
}

void try_to_change(int x, int y){
    x = x + 10;
    y = y + 10;
    printf("\nValue of x (inside function) = %d\n", x);
    printf("Value of y (inside function) = %d\n", y);
}
```

Copy the value from main's x and y to try_to_change's x and y

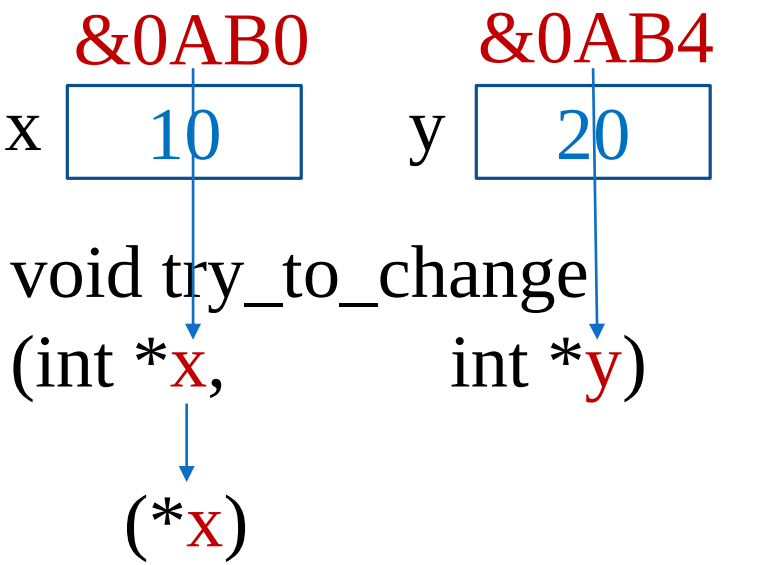
LECTURE 15

Call by Reference

Call by Reference in C

mimicked by passing pointer value

- The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call.
- It means the changes made to the parameter affect the passed argument.
- To pass a value by reference, argument pointers are passed to the functions just like any other value.



Note:
x and x are different

```c
#include<stdio.h>
void try_to_change(int *, int *);

int main(){
   int x = 10, y = 20;
   printf("Initial value of x = %d\n", x);
   printf("Initial value of y = %d\n", y);
   printf("\nCalling the function\n");
   try_to_change(&x, &y);
   printf("\nValues after function call\n\n");
   printf("Final value of x = %d\n", x);
   printf("Final value of y = %d\n", y);
   // signal to operating system everything works fine
   return 0;
}

void try_to_change(int *x, int *y){
   (*x)++;
   (*y)++;
   printf("\nValue of x (inside function) = %d\n", *x);
   printf("Value of y (inside function) = %d\n", *y);
}
```

int *x => read as x's body is an integer;

Demo Program:

callByReference.c

Go gcc!!!