C for Java programmers

With Thanks to
Indranil Gupta &
Cornell

Why learn C (after Java)?

- Both high-level and low-level language
- Better control of low-level mechanisms
- Performance better than Java (Unix, NT !)
- Java hides many details needed for writing OS code

  But,….
- Memory management responsibility
- Explicit initialization and error detection
- More room for mistakes

What does this C program do ?

```
#include <stdio.h>
struct list{int data; struct list *next};
struct list *start, *end;
void add(struct list *head, struct list *list, int data};
int delete(struct list *head, struct list *tail);

void main(void){
 start=end=NULL;
 add(start, end, 2);    add(start, end, 3);
 printf(“First element: %d”, delete(start, end));
}

void add(struct list *head, struct list *tail, int data}{
 if(tail==NULL){
  head=tail=malloc(sizeof(struct list));
  head->data=data; head->next=NULL;
 }
 else{
  tail->next= malloc(sizeof(struct list));
  tail=tail->next; tail->data=data; tail->next=NULL;
 }
}
```

```
void delete (struct list *head, struct list *tail){
 struct list *temp;
 if(head==tail){
  free(head); head=tail=NULL;
 }
 else{
  temp=head->next; free(head); head=temp;
 }
}
```

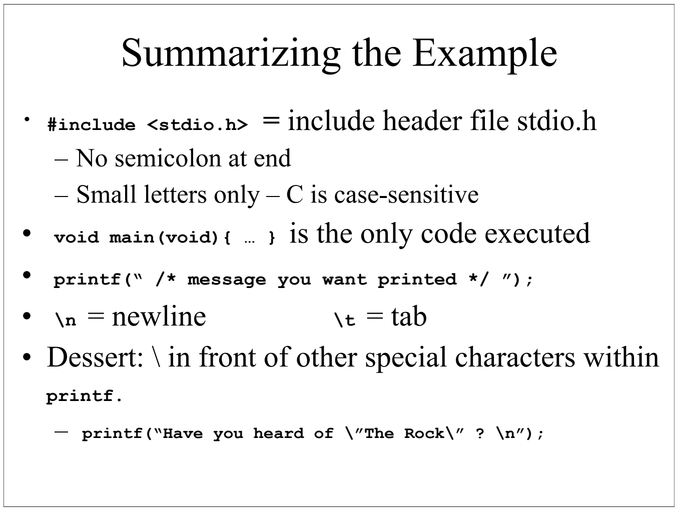
Goals of this tutorial

- To introduce some basic C concepts to you
  - so that you can read further details on your own
- To warn you about common mistakes made by beginners
  - so that you get your homework done quickly
- You will be able to understand the earlier complicated program completely !
  - And write more complicated code

Simple Example

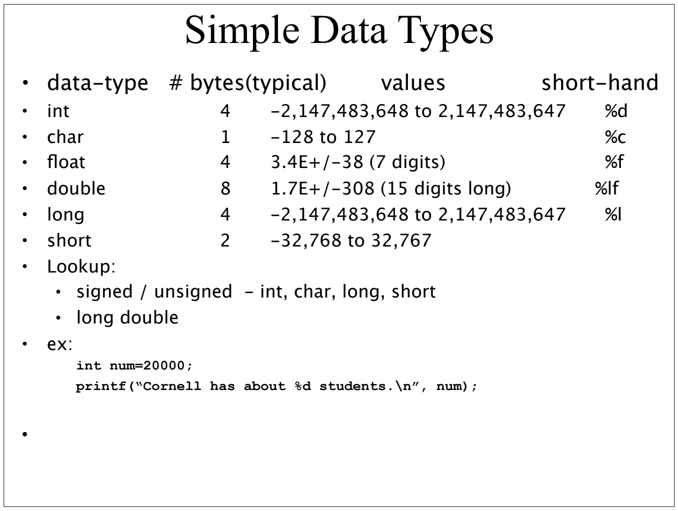
```
#include <stdio.h>

void main(void)
{
    printf(“Hello World. \n \t and you ! \n ”);
            /* print out a message */
    return;
}
```

```
$Hello World.
        and you !
$
```



Summarizing the Example

- `#include <stdio.h>` = include header file stdio.h
  - No semicolon at end
  - Small letters only – C is case-sensitive
- `void main(void){ … }` is the only code executed
- `printf(“ /* message you want printed */ ”);`
- `\n` = newline `\t` = tab
- Dessert: \ in front of other special characters within `printf`.
  - `printf(“Have you heard of \”The Rock\” ? \n”);`

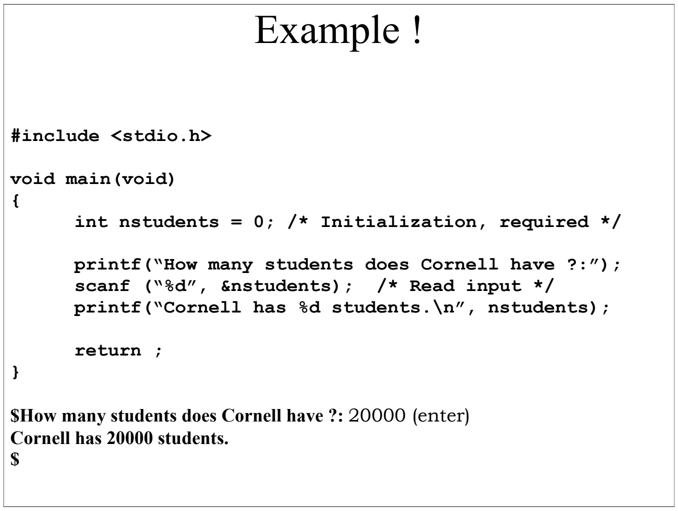


Simple Data Types

data-type	# bytes(typical)	values	short-hand
int	4	-2,147,483,648 to 2,147,483,647	%d
char	1	-128 to 127	%c
float	4	3.4E+/-38 (7 digits)	%f
double	8	1.7E+/-308 (15 digits long)	%lf
long	4	-2,147,483,648 to 2,147,483,647	%l
short	2	-32,768 to 32,767	

- Lookup:
  - signed / unsigned - int, char, long, short
  - long double
- ex:

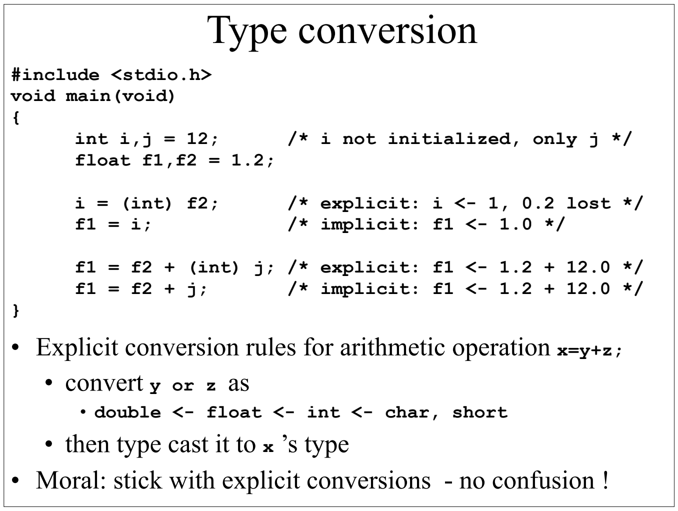
```
int num=20000;
printf(“Cornell has about %d students.\n”, num);
```



Example !

```
#include <stdio.h>

void main(void)
{
      int nstudents = 0; /* Initialization, required */

      printf(“How many students does Cornell have ?:”);
      scanf (“%d”, &nstudents);  /* Read input */
      printf(“Cornell has %d students.\n”, nstudents);

      return ;
}
```

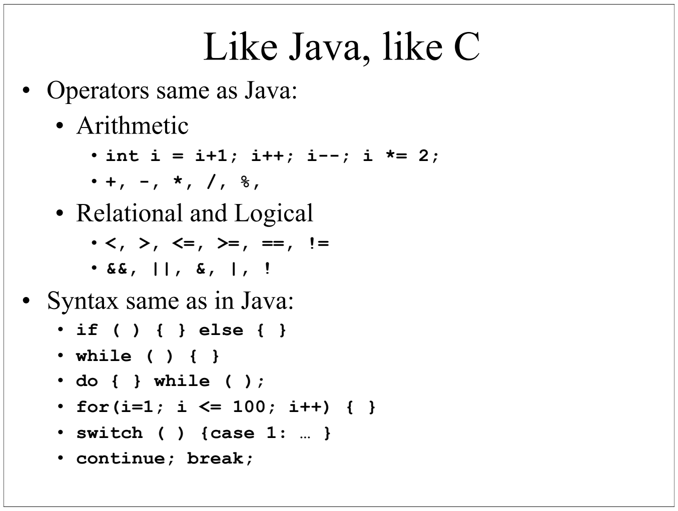
```
$How many students does Cornell have ?: 20000 (enter)
Cornell has 20000 students.
$
```



Type conversion

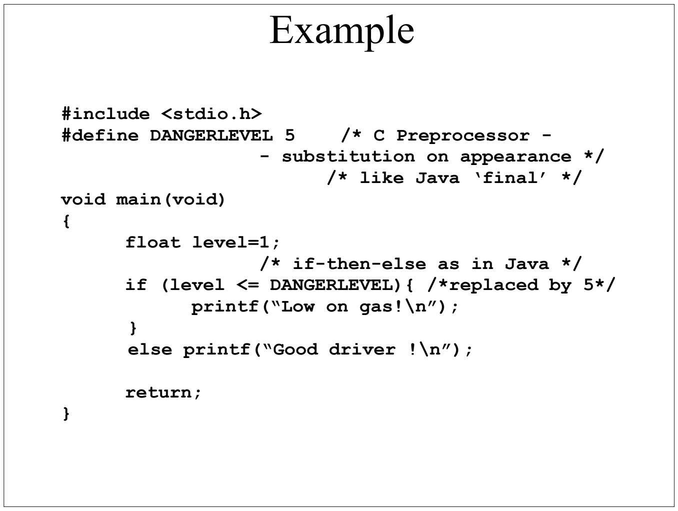
```
#include <stdio.h>
void main(void)
{
      int i,j = 12;      /* i not initialized, only j */
      float f1,f2 = 1.2;

      i = (int) f2;      /* explicit: i <- 1, 0.2 lost */
      f1 = i;            /* implicit: f1 <- 1.0 */

      f1 = f2 + (int) j; /* explicit: f1 <- 1.2 + 12.0 */
      f1 = f2 + j;       /* implicit: f1 <- 1.2 + 12.0 */
}
```

- Explicit conversion rules for arithmetic operation `x=y+z;`
  - convert `y or z` as
    - `double <- float <- int <- char, short`
  - then type cast it to `x` ’s type
- Moral: stick with explicit conversions - no confusion !



Like Java, like C

- Operators same as Java:
  - Arithmetic
    - `int i = i+1; i++; i--; i *= 2;`
    - `+, -, *, /, %,`
  - Relational and Logical
    - `<, >, <=, >=, ==, !=`
    - `&&, ||, &, |, !`
- Syntax same as in Java:
  - `if ( ) { } else { }`
  - `while ( ) { }`
  - `do { } while ( );`
  - `for(i=1; i <= 100; i++) { }`
  - `switch ( ) {case 1: … }`
  - `continue; break;`



Example

```
#include <stdio.h>
#define DANGERLEVEL 5    /* C Preprocessor -
                 - substitution on appearance */
                       /* like Java ‘final’ */
void main(void)
{
      float level=1;
                  /* if-then-else as in Java */
      if (level <= DANGERLEVEL){ /*replaced by 5*/
            printf(“Low on gas!\n”);
      }
       else printf(“Good driver !\n”);

      return;
}
```

One-Dimensional Arrays

```
#include <stdio.h>

void main(void)
{
    int number[12]; /* 12 cells, one cell per student */
    int index, sum = 0;
            /* Always initialize array before use */
    for (index = 0; index < 12; index++) {
      number[index] = index;
    }
    /* now, number[index]=index; will cause error:why ?*/

    for (index = 0; index < 12; index = index + 1) {
      sum += number[index];  /* sum array elements */
    }

    return;
}
```

More arrays

- Strings

```
char name[6];
name = {'C','S','4','1','4','\0'};
                                        /* '\0'=
  end of string */
printf("%s", name); /* print until '\0' */
```

  - Functions to operate on strings
    - `strcpy, strncpy, strcmp, strncmp, strcat, strncat, strstr,strchr`
    - `#include <strings.h>` at program start

- Multi-dimensional arrays

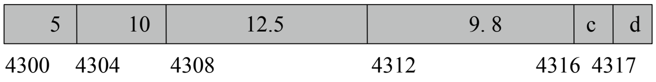
```
int points[3][4];
points [1][3] = 12;  /* NOT points[3,4] */
printf("%d", points[1][3]);
```

Like Java, somewhat like C

- Type conversions
  - but you can typecast from any type to any type
    - `c = (char) some_int;`
  - So be careful !
- Arrays
  - *Always* initialize before use
  - `int number[12];`
    `printf("%d", number[20]);`
    - produces undefined output, may terminate, may not even be detected.

Memory layout and addresses

```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

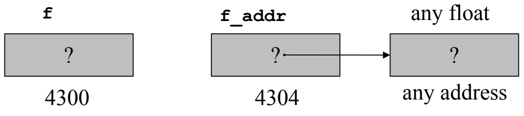


5	10	12.5	9. 8	c	d
4300	4304	4308	4312	4316	4317

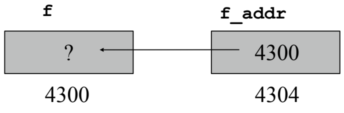
Pointers made easy - 1

- *Pointer* = variable containing address of another variable

```
float f;        /* data variable */
float *f_addr;  /* pointer variable */
```

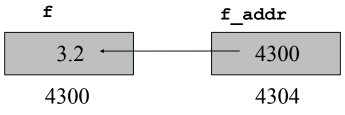


```
f_addr = &f; /* & = address operator */
```

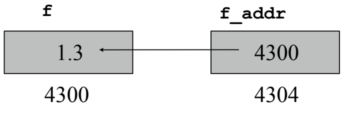


Pointers made easy - 2

```
*f_addr = 3.2;    /* indirection operator */
```



```
float g=*f_addr; /* indirection:g is now 3.2 */

f = 1.3;
```



Pointer Example

```
#include <stdio.h>

void main(void) {
      int j;
      int *ptr;

      ptr=&j;   /* initialize ptr before using it */
                /* *ptr=4 does NOT initialize ptr */

      *ptr=4;     /* j <- 4 */

      j=*ptr;     /* j <- ??? */
}
```

Dynamic Memory allocation

- Explicit allocation and de-allocation

```
#include <stdio.h>

void main(void) {
      int *ptr;
             /* allocate space to hold an int */
      ptr = malloc(sizeof(int));

            /* do stuff with the space */
      *ptr=4;

      free(ptr);
            /* free up the allocated space */
}
```

Elementary file handling

```
#include <stdio.h>

void main(void) {
                         /* file handles */
    FILE *input_file=NULL;

                        /* open files for writing*/
    input_file = fopen(“cwork.dat”, “w”);
    if(input_file == NULL)
            exit(1);    /* need to do explicit ERROR
   CHECKING */

                  /* write some data into the file */
    fprintf(input_file, “Hello there”);

                  /* don’t forget to close file handles */
    fclose(input_file);

    return;
}
```

Error Handling

- Moral from example:
  - unlike Java, no explicit exceptions
  - need to manually check for errors
    - Whenever using a function you’ve not written
    - Anywhere else errors might occur

Functions - why and how ?

- If a program is too long
- Modularization – easier to
  - code
  - debug
- Code reuse
- In C, everything is CALL BY VALUE!!!
- Passing arguments to functions
  - By value
  - By reference
- Returning values from functions
  - By value
  - By reference

Functions – basic example

```
#include <stdio.h>
int sum(int a, int b);
            /* function prototype at start of file */

void main(void){
   int total = sum(4,5); /* call to the function */

   printf(“The sum of 4 and 5 is %d”, total);
}

int sum(int a, int b){    /* the function itself
                        - arguments passed by value*/
      return (a+b);      /* return by value */
}
```

Arguments by reference

```
#include <stdio.h>
int sum(int *pa, int *pb);
            /* function prototype at start of file */

void main(void){
   int a=4, b=5;
   int *ptr = &b;
   int total = sum(&a,ptr); /* call to the function */

   printf(“The sum of 4 and 5 is %d”, total);
}

int sum(int *pa, int *pb){   /* the function itself
                        - arguments passed by reference */
      return (*pa+*pb);      /* return by value */
}
```

Why pointer arguments?!

```
#include <stdio.h>

void swap(int, int);

main() {
  int num1 = 5, num2 = 10;
  swap(num1, num2);
  printf(“num1 = %d and num2 = %d\n”, num1, num2);
}

void swap(int n1, int n2) { /* passed by value */
  int temp;

  temp = n1;
  n1 = n2;
  n2 = temp;
}
```

Why pointer arguments? This is why

```
#include <stdio.h>

void swap(int *, int *);

main() {
  int num1 = 5, num2 = 10;
  swap(&num1, &num2);
  printf(“num1 = %d and num2 = %d\n”, num1, num2);
}

void swap(int *n1, int *n2) { /* passed and returned by
                                    reference */
  int temp;

  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

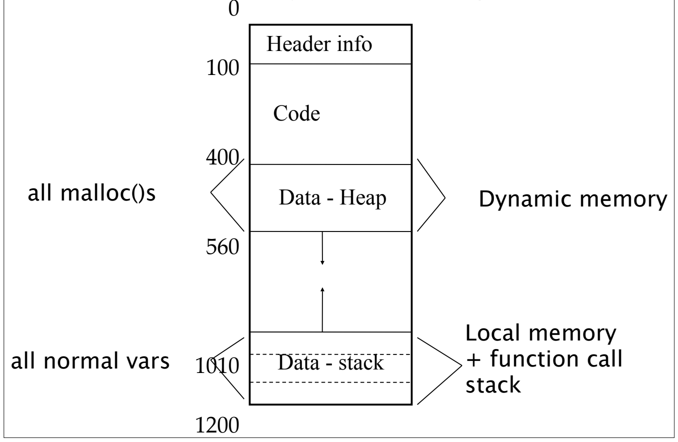
What’s wrong with this ?

```
#include <stdio.h>

void dosomething(int *ptr);

main() {
      int *p;
      dosomething(p)
      printf(“%d”, *p);     /* will this work ? */
}

void dosomething(int *ptr){ /* passed and returned by
                                    reference */
  int temp=32+12;

  ptr = &(temp);
}

/* compiles correctly, but gives run-time error */
```

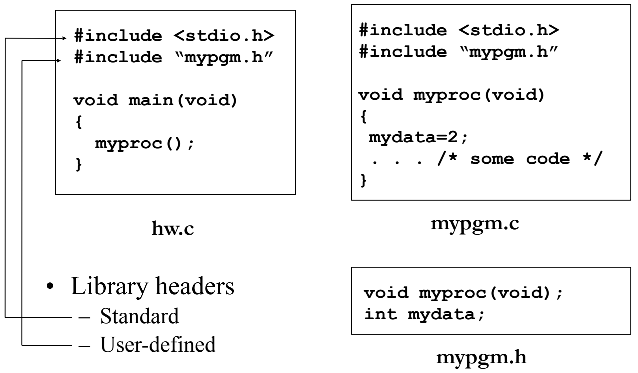
Passing and returning arrays

```
#include <stdio.h>

void init_array(int array[], int size) ;

void main(void) {
  int list[5];

  init_array(list, 5);
  for (i = 0; i < 5; i++)
    printf(“next:%d”, array[i]);
}

void init_array(int array[], int size) { /* why size ? */
             /* arrays ALWAYS passed by reference */
 int i;
 for (i = 0; i < size; i++)
    array[i] = 0;
}
```

Memory layout of programs



Program with multiple files



```
#include <stdio.h>
#include “mypgm.h”

void main(void)
{
  myproc();
}
```

hw.c

```
#include <stdio.h>
#include “mypgm.h”

void myproc(void)
{
 mydata=2;
 . . . /* some code */
}
```

mypgm.c

- Library headers
  - Standard
  - User-defined

```
void myproc(void);
int mydata;
```

mypgm.h

Externs

```
#include <stdio.h>

extern char user2line [20];   /* global variable defined
                                 in another file */
char user1line[30];           /* global for this file */
void dummy(void);

void main(void) {
 char user1line[20];       /* different from earlier
                                   user1line[30] */
 . . .                    /* restricted to this func */
}

void dummy(){
 extern char user1line[];  /* the global user1line[30] */
 . . .
}
```

Structures

- Equivalent of Java’s classes with only data (no methods)

```
#include <stdio.h>

struct birthday{
    int month;
    int day;
    int year;
  };

main() {
  struct birthday mybday;     /* - no ‘new’ needed ! */
                          /* then, it’s just like Java ! */
  mybday.day=1; mybday.month=1; mybday.year=1977;
  printf(“I was born on %d/%d/%d”, birth.day,
                            birth.month, birth.year);
}
```

More on Structures

```
struct person{
    char name[41];
    int age;
    float height;
    struct {         /* embedded structure */
      int month;
      int day;
      int year;
    } birth;
 };

struct person me;

me.birth.year=1977;………

struct person class[60];
      /* array of info about everyone in class */

class[0].name=“Gun”; class[0].birth.year=1971;……
```

Passing/Returning a structure

```
            /* pass struct by value */
void display_year_1(struct birthday mybday) {
  printf(“I was born in %d\n”, mybday.year);
}                 /* - inefficient: why ? */
. . . .
            /* pass struct by reference */
void display_year_2(struct birthday *pmybday) {
  printf(“I was born in %d\n”, pmybday->year);
      /* warning ! ‘->’, not ‘.’, after a struct pointer*/
}
. . . .
            /* return struct by value */
struct birthday get_bday(void){
  struct birthday newbday;
  newbday.year=1971;  /* ‘.’ after a struct */
  return newbday;
}                 /* - also inefficient: why ? */
```

enum - enumerated data types

```
#include <stdio.h>
enum month{
      JANUARY,      /* like #define JANUARY 0 */
      FEBRUARY,     /* like #define FEBRUARY 1 */
      MARCH         /* … */
};

/* JANUARY is the same as month.JANUARY */

/* alternatively, …. */

enum month{
      JANUARY=1,    /* like #define JANUARY 1 */
      FEBRUARY,     /* like #define FEBRUARY 2 */
      MARCH         /* … */
};
```

Synonym for a data type

```
typedef int Employees;

Employees my_company;  /* same as int my_company; */

typedef struct person Person;

Person me;             /* same as struct person me; */

typedef struct person *Personptr;

Personptr ptrtome;     /* same as struct person *ptrtome;*/
```

- Easier to remember
- Clean code

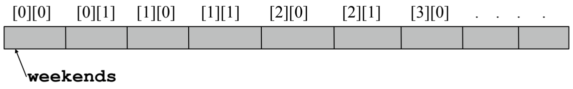
More pointers

```
int month[12]; /* month is a pointer to base address 430*/

month[3] = 7;  /* month address + 3 * int elements
              => int at address (430+3*4) is now 7 */

ptr = month + 2; /* ptr points to month[2],
              => ptr is now (430+2 * int elements)= 438 */
ptr[5] = 12;
               /* ptr address + 5 int elements
              => int at address (434+5*4) is now 12.
                   Thus, month[7] is now 12 */

ptr++;         /* ptr <- 438 + 1 * size of int = 442 */
(ptr + 4)[2] = 12;   /* accessing ptr[6] i.e., array[9] */
```

- Now , `month[6], *(month+6), (month+4)[2], ptr[3], *(ptr+3)` are all the same integer variable.

2-D arrays

- 2-dimensional array
  `int weekends[52][2];`



- `weekends[2][1]` is same as `*(weekends+2*2+1)`
  - NOT `*weekends+2*2+1` :this is an int !

Pointer Example - argc and argv parameters

```
#include <stdio.h>
        /* program called with cmd line parameters */

void main(int argc, char *argv[]) {
  int ctr;

  for (ctr = 0; ctr < argc; ctr = ctr + 1) {
    printf(“Argument #%d is -> |%s|\n”, ctr, argv[ctr]);
  }     /* ex., argv[0] == the name of the program */
}
```

Strings

```
#include <stdio.h>

main() {
  char msg[10]; /* array of 10 chars */
  char *p;      /* pointer to a char */
  char msg2[]=“Hello”; /* msg2 = ‘H’’e’’l’’l’’o’’\0’ */

  msg = “Bonjour”; /* ERROR. msg has a const address.*/
  p   = “Bonjour”; /* address of “Bonjour” goes into p */

  msg = p; /* ERROR. Message has a constant address. */
           /* cannot change it. */

  p = msg; /* OK */

  p[0] = ‘H’, p[1] = ‘i’,p[2]=‘\0’;
           /* *p and msg are now “Hi” */
}
```

Pointer to function

```
int func(); /*function returning integer*/
int *func(); /*function returning pointer to integer*/
int (*func)(); /*pointer to function returning integer*/
int *(*func)(); /*pointer to func returning ptr to int*/
```

- Advantage ? more flexibility

Pointer to function - Example

```
#include <stdio.h>

void myproc (int d);
void mycaller(void (* f)(int), int param);

void main(void) {
      myproc(10);       /* call myproc with parameter 10*/
      mycaller(myproc, 10); /* and do the same again ! */
}

void mycaller(void (* f)(int), int param){
      (*f)(param);      /* call function *f with param */
}

void myproc (int d){
      . . .             /* do something with d */
}
```

Doing more complicated things…

```
To declare an array of N pointers to functions returning
pointers to functions returning pointers to characters

1. char *(*(*a[N])())();

2. Build the declaration up in stages, using typedefs:

typedef char *pc; /* pointer to char */
typedef pc fpc(); /* function returning pointer to char */
typedef fpc *pfpc; /* pointer to above */
typedef pfpc fpfpc(); /* function returning... */
typedef fpfpc *pfpfpc; /* pointer to... */
pfpfpc a[N]; /* array of... */
```

What does this C program do ?

```
#include <stdio.h>
struct list{int data; struct list *next};
struct list *start, *end;
void add(struct list *head, struct list *list, int data};
int delete(struct list *head, struct list *tail);

void main(void){
 start=end=NULL;
 add(start, end, 2);    add(start, end, 3);
 printf(“First element: %d”, delete(start, end));
}

void add(struct list *head, struct list *tail, int data}{
 if(tail==NULL){
  head=tail=malloc(sizeof(struct list));
  head->data=data; head->next=NULL;
 }
 else{
  tail->next= malloc(sizeof(struct list));
  tail=tail->next; tail->data=data; tail->next=NULL;
 }
}
```

```
void delete (struct list *head, struct list *tail){
 struct list *temp;
 if(head==tail){
  free(head); head=tail=NULL;
 }
 else{
  temp=head->next; free(head); head=temp;
 }
}
```

Before you go….

- Always initialize anything before using it (especially pointers)
- Don’t use pointers after freeing them
- Don’t return a function’s local variables by reference
- No exceptions – so check for errors everywhere
- An array is also a pointer, but its value is immutable.