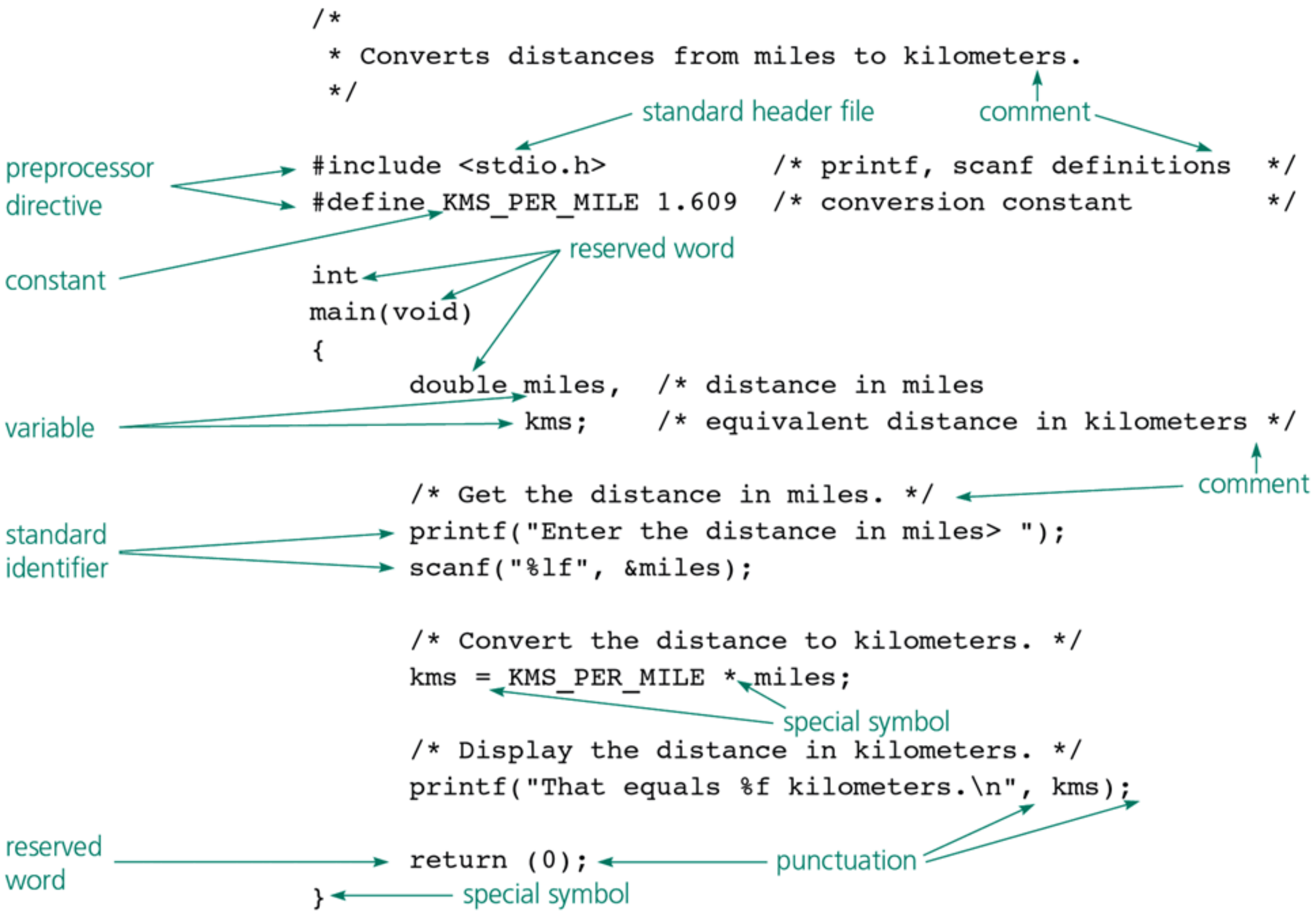
Software Implementation and Algorithm Design

Interactive Data Input & Output

- a mode of program execution in which the user responds to prompts by entering data

Interactive Data Input & Output



```
/*
 * Converts distances from miles to kilometers.
 */

#include <stdio.h>          /* printf, scanf definitions  */
#define KMS_PER_MILE 1.609  /* conversion constant        */

int
main(void)
{
      double miles,  /* distance in miles
             kms;    /* equivalent distance in kilometers */

      /* Get the distance in miles. */
      printf("Enter the distance in miles> ");
      scanf("%lf", &miles);

      /* Convert the distance to kilometers. */
      kms = KMS_PER_MILE * miles;

      /* Display the distance in kilometers. */
      printf("That equals %f kilometers.\n", kms);

      return (0);
}
```

Scope Rules

- Local Variables
  - Declared at the beginning of functions
  - Scope is the function body
- Global Variables
  - Declared outside functions
  - Scope is from the point where they are declared until end of file (unless prefixed by extern which will be discussed later)

Scope Rules

- Variables can be declared within blocks too
  - scope is until end of the block

```
{
    int block_variable;
}
block_variable = 9; (wrong)
```

Global and Local Variables

global-var.c

```
#include <stdio.h>
int x;      // global variable
int main() {
   x=10;
   printf(“x=%d\n”,x);
   printf(“x=%d\n”,fun1());
   printf(“x=%d\n”,fun2());
   printf(“x=%d\n”,fun3());   }

int fun1()  {
   x=x+10;
   return(x);   }

int fun2()  {
   int x;      // local variable
   x=1;
   return(x);   }

int fun3()   {
   x=x+10;
   return(x);   }

```

Global and Local Variables

- Once a variable has been declared global, any function can use it and change its value.

Global Variables

```
sepc92:> gcc global-var.c –o global-var
sepc92:> ./global-var                     ... run the program
x=10
x=20
x=1
x=30
```

Structure in C

- Computing applications often need data of different kinds to be associated. A typical place where this occurs is in a *record*.
- For example a database record for a particular student might contain the student number, full name, course-code and year of entry.

Record Declaration using Structure

```
struct student {
        char studno[12];
        char name[50];
        char course[7];
        int startyear;
};
```

More Examples

```
struct coord { float x,y; };
struct complex { float real,imaginary; };
```

The data might have different types but does not have to be of different types. Structures also help with the *naming* of related parts of a record object.

Allocating Memory to Records

- The previous declarations don't create any space for data storage, but they provide a template for data of the appropriate types to be allocated.
- Let's allocate some storage to data variables:

```
struct coord origin, cursor;
  /* 2 x-y coordinates called origin and cursor */
struct student sdn_year2[50];
  /* array for 50 students */
```

Defining a New Data Type

- Better to have one word e.g. **COORD** to declare the type of data than 2 words

  e.g. **struct coord**

- This needs a *typedef* statement:

```
struct coord { float x,y; };
typedef struct coord COORD;
/* COORD now same as struct coord */
COORD origin, cursor;
/* declare 2 x-y coordinates called origin and cursor */
```

dot structure member access operator

Members x and y of origin and cursor can now be accessed as origin.x, origin.y

cursor.x and cursor.y e.g.

```
origin.x=3.5;
origin.y=2.0;
```

Functions with Structured Parameters

```
float distance(COORD a, COORD b){
    /* calculate distance between a and b */
     float z,vert,horiz;
    /* z = distance, vert = y1 - y2,   horiz = x1 - x2 */
     horiz=a.x - b.x; /* the horizontal distance */
     vert=a.y - b.y;   /* the vertical distance */
    /* calculate z as the hypotenuse of a right angle triangle */
     z=sqrt((horiz*horiz) + (vert*vert));
                            /* Pythagorus theorem: */
     return z;              /* z*z = x*x + y*y    */
}
```

Functions with Structured Return Values

```
COORD getcoord(void) {
        /* note returned type is COORD */
       /* prompts for and returns a coordinate */
         COORD temp;
         printf("Enter x and y  coordinates \n");
         scanf("%f%f", &temp.x, &temp.y);
         return temp;
}
```

A Complete Source Program Part 1

```
/*  coord.c   */
#include <stdio.h>
#include <math.h>
   /* needed to use sqrt() square root */

struct coord { float x,y; };
  /* declare structure coord as having x and y members */
typedef struct coord COORD;
  /* COORD is now same as struct coord */

/* function prototypes */
float distance(COORD a, COORD b);
COORD getcoord(void);
```

A Complete Source Program Part 2

```
int main(void){
        COORD origin, cursor;
        float separation;
        printf("enter details for origin:\n");
        origin=getcoord();
        printf("enter details for cursor:\n");
        cursor=getcoord();
        separation=distance(origin,cursor);
        printf("the distance between origin and cursor is %f\n",
                                                        separation);
        return 0;
}
```

A Complete Source Program Part 3

```
float distance(COORD a, COORD b)
{
      :
      :
}

COORD getcoord(void)
{
      :
      :
}
```

Compile and Execute

```
sepc92:> gcc coord.c –lm
```

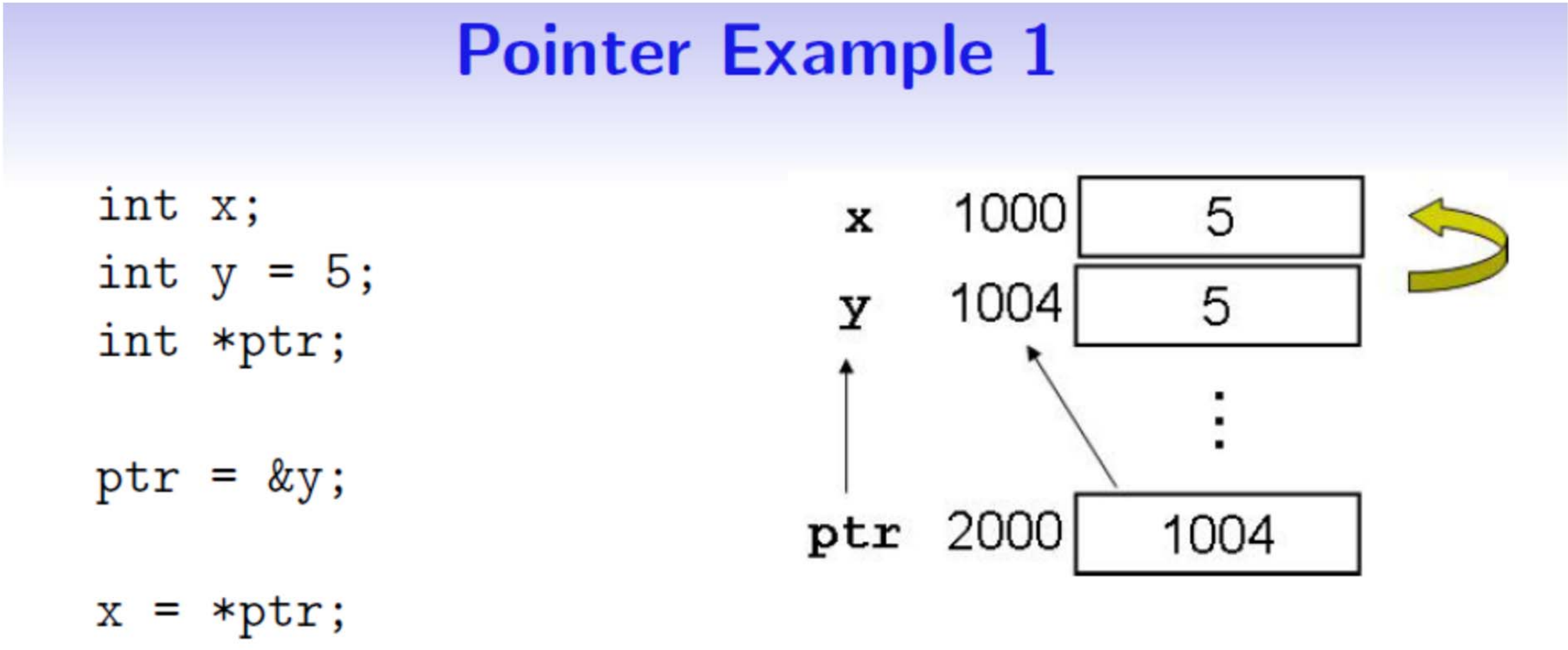
- The *–lm* means “linking with the math library”. It is needed because of *#include <math.h>*

```
sepc92:> ./a.out                              … run the program
enter details for origin:
Enter x and y coordinates
1  1
enter details for origin:
Enter x and y coordinates
3  2
the distance between origin and cursor is 2.236068
```

Pointer Variables

- Pointer variables in C are variables that store memory addresses.
- Pointer declaration:
  - int x, y = 5;
  - int *ptr;
  - /*ptr is a POINTER to an integer variable*/
- Reference operator &:
  - ptr = &y;
  - /*assign ptr to the MEMORY ADDRESS of y.*/
- Dereference operator *:
  - x = *ptr;
  - /*assign x to the int that is pointed to by ptr */

Pointer Variables



Pointer Example 1

```
int x;
int y = 5;
int *ptr;

ptr = &y;

x = *ptr;
```

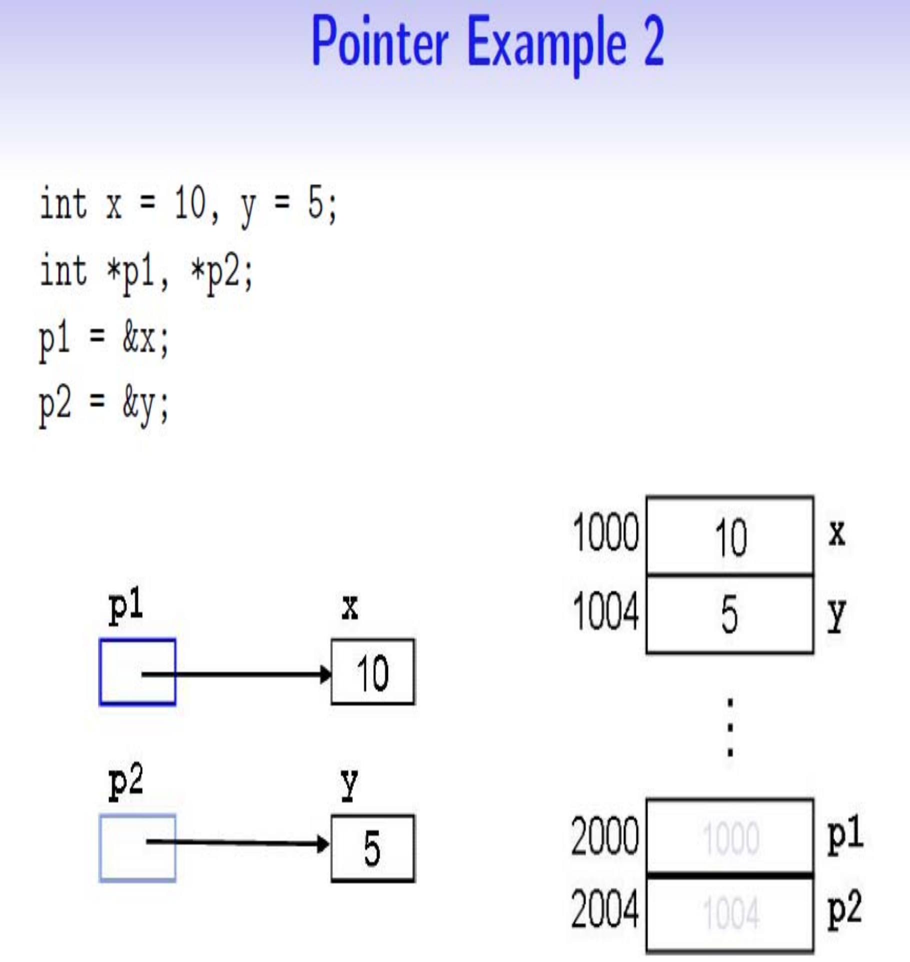
x 1000 5

y 1004 5

⋮

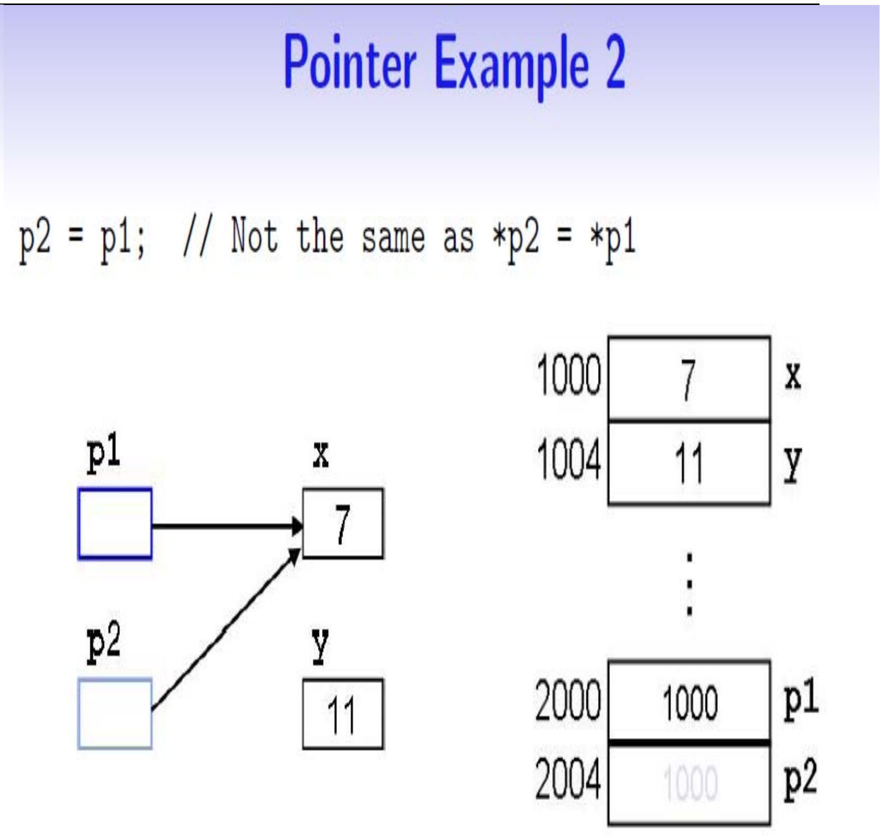
ptr 2000 1004

Pointer Variables



Pointer Example 2

```
int x = 10, y = 5;
int *p1, *p2;
p1 = &x;
p2 = &y;
```



Pointer Example 2

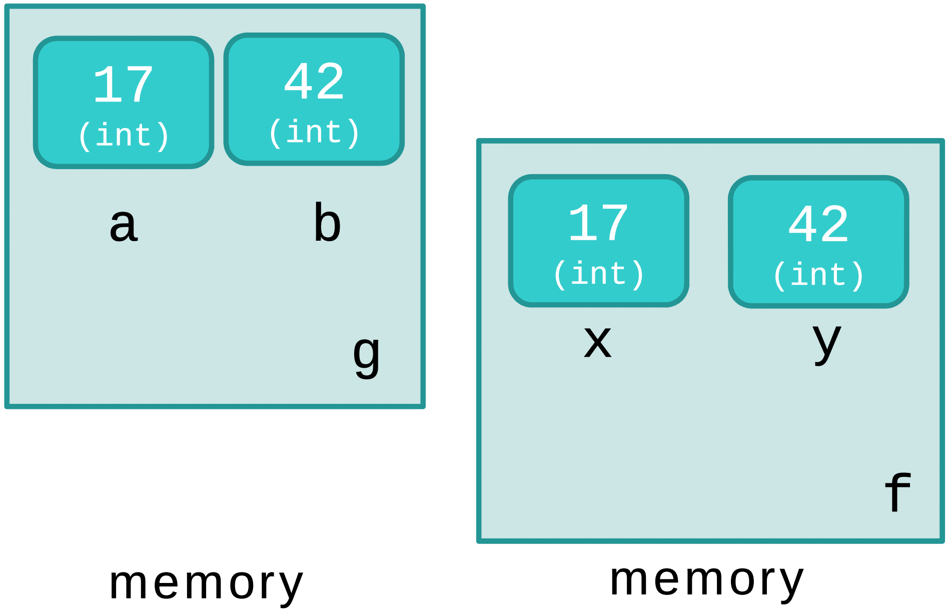
```
p2 = p1;  // Not the same as *p2 = *p1
```

Function Arguments

- In C, arguments are passed “by value”
  - A temporary copy of each argument is created, solely for use within the function call

```
void f(int x, int y) {
      :
  z = x + y;
      :
}

void g(…) {
  int a = 17, b = 42;
  f(a, b);
  …
}
```



Function Arguments – swapattempt()

```
/* swapattempt.c  - attempt to swap two values through
   function parameters */
#include <stdio.h>
void swapattempt(int x, int y);
int main()  {
   int x = 10;
   int y = 20;
   printf("before swapattempt: x=%d y=%d\n",x,y);
   swapattempt(x,y);
   printf("after swapattempt: x=%d y=%d\n",x,y);
   return 0;
}
void swapattempt(int x,int y)  {
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Function Arguments – swapattempt()

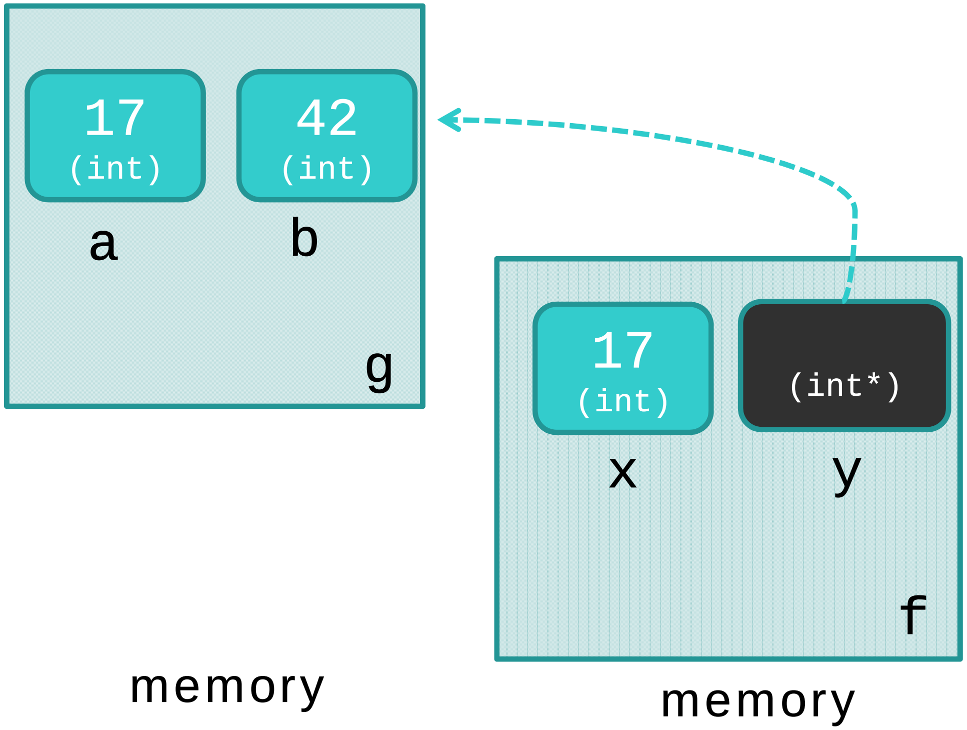
memory address

	2560	2561	5036	5037
	x in main()	y in main()	x in swapattempt()	y in swapattempt()
In main(): Just before calling swapattempt(x,y)	10	20		
In swapattempt(): Just before the first code			10	20
In swapattempt(): Just after y=temp			20	10
In main(): Just after calling swapattempt(x,y)				

Function Arguments with Pointers

```
void f(int x, int *y) {
      :
  z = x + *y;
      :
}

void g(…) {
  int a = 17, b = 42;
  f(a, &b);
  …
}
```



Function Arguments

```c
/* swap.c -   swap two values through function parameters */
#include <stdio.h>
void swap(int* iPtrX, int *iPtrY);
int main()  {
   int x = 10;
   int y = 20;
   int* p1 = &x;
   int *p2 = &y;
   printf("before swap: x=%d y=%d\n",x,y);
   swap(p1,p2);
   printf("after swap: x=%d y=%d\n",x,y);
   return 0;
}
void swap(int* iPtrX, int *iPtrY) {
   int temp;
   temp = *iPtrX;
   *iPtrX = *iPtrY;
   *iPtrY = temp;
}
```

Function Arguments

memory address

	2560	2561	2562	2563	5036	5037	5038
	x in main()	y in main()	p1 in main()	p2 in main()	iPtrX in swap()	iPtrY in swap()	temp in swap()
In main(): Just before calling swap(p1,p2)	10	20	2560	2561			
In swap(): Just before the first code					2560	2561	garbage
In swap(): Just after temp = *iPtrX							10
In swap(): Just after *iPtrX = *iPtrY	20						
In swap(): Just after *iPtrY = temp		10					
In main(): Just after calling swap(p1,p2)							

Use of Structure Pointers

- Consider the structure COORD in the previous example. Suppose **a** and **b** are of type COORD
- Members **x** and **y** of coordinates **a** and **b** can also be accessed through pointers to **a** and **b** so that if pointer **p** stores the address of a: **p=&a;** then

p->x

directly accesses member x of a or **a.x** .

Rewrite the Above Program Using Pointers

```
/* declarations above here of headers , struct coord and
typedef COORD same as before so not repeated */

float distance(COORD *a, COORD *b);

/* a and b are now pointers to COORD */

void getcoord(COORD *t);

/* inputs coordinate through COORD* pointer t */
```

Rewrite Using Pointers - Part 2

```
int main(void)  {
        COORD origin, cursor, *orig,*curs;
        orig=&origin; curs=&cursor;
        /* store addresses in pointers orig and curs */
        float separation;
        printf("enter details for origin:\n");
        getcoord(orig);
        printf("enter details for cursor:\n");
        getcoord(curs);
        separation=distance(orig,curs);
        printf("the distance between origin and cursor %f\n" ,
                                                            separation);
        return 0;
}
```

Rewrite Using Pointers - Part 3

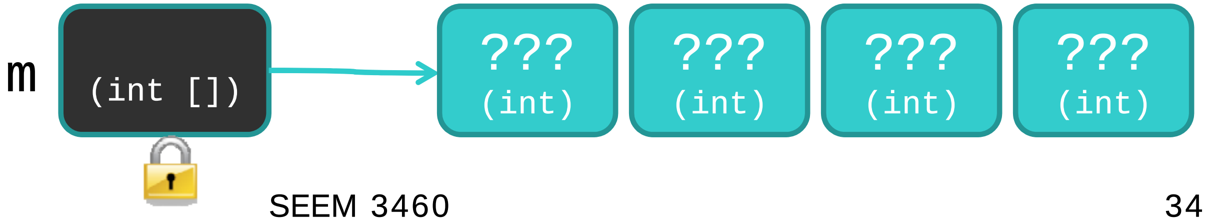
```
float distance(COORD *a, COORD *b)  {
  /* calculate distance between a and b */
  float z,vert,horiz;  /* z = distance, vert = y1-y2,  horiz = x1 - x2 */
  horiz=a->x - b->x; /* horizontal distance note -> pointer syntax */
  vert=a->y - b->y;   /* the vertical distance */
      /* calculate z as the hypotenuese of a right angle triangle */
  z=sqrt((horiz*horiz) + (vert*vert)); /* pythagorus theorem: */
      return z;                          /* z*z = x*x + y*y     */
}

void getcoord(COORD *t){ /* inputs x-y coordinate using pointers */
    printf("please enter x and y coordinates\n");
    scanf("%f%f",&t->x,&t->y);
                        /* -> has higher precedence than &  */
}
```

Review of Arrays

- There are no array variables in C – only array *names*
  - Each name refers to a constant pointer
  - Space for array elements is allocated at declaration time
- Can’t change where the array name refers to…
  - but you can change the array elements, via pointer arithmetic

```
int m[4];
```



Subscripts and Pointer Arithmetic

- `array[subscript]` equivalent to `*(array + (subscript))`

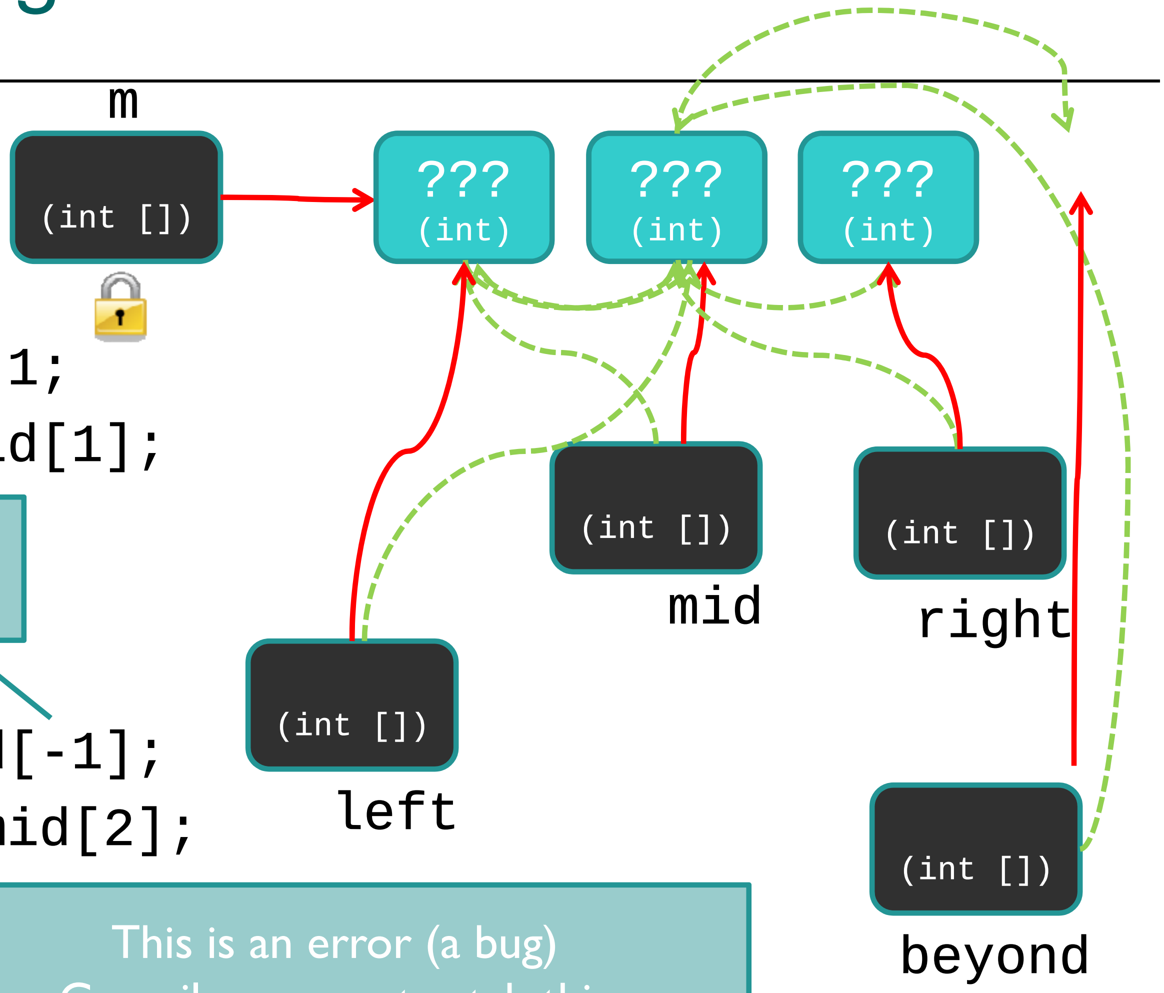
Array Names and Pointer Variables - Playing Together

```
int m[3];

int *mid = m + 1;
int *right = mid[1];
```

subscript OK with pointer variable

```
int *left = mid[-1];
int *beyond = mid[2];
```



This is an error (a bug)
Compiler may not catch this
Execution may get strange results

Array Names as Function Arguments

- If a function argument is an array name, it is also passed by value.
- But only the pointer is copied, not the array elements!
  - But – the pointer copy points to the same elements as the callee's array
  - The array elements can be modified inside the function
- Array declarations in function parameter need not have a size specified

```
/*  reverse.c  */
#include <stdio.h>
#include <string.h>
/*  Function prototype  */
void reverse (char before[], char after[]);

/*******************************************/
int main()
{
  char str[100];   /* buffer to hold reversed string */
  reverse("cat",str);  /* reverse the string "cat"  */
  printf ("reverse(\"cat\") = %s\n", str);
  reverse("noon",str);
  printf ("reverse(\"noon\") = %s\n", str);
}

/*******************************************/
void reverse (char before[], char after[])
{
  int i,j,len;

  len = strlen(before);
  i=0;
  for (j=len-1; j>=0; j--)
  {
    after[i] = before[j];
    i++;
  }
  after[len] = ‘\0’;
}
```

Array Argument Example

- The memory storage for the arrays *before* and *after* is in the main function.
- The function arguments just pass the array name
- Combined with pass by value in C, the final effect is that it passes the pointer corresponding to the array names

When Can Array Size be Omitted?

- There are a couple of contexts in which an array declaration need not have a size specified:
  - Parameter declaration:

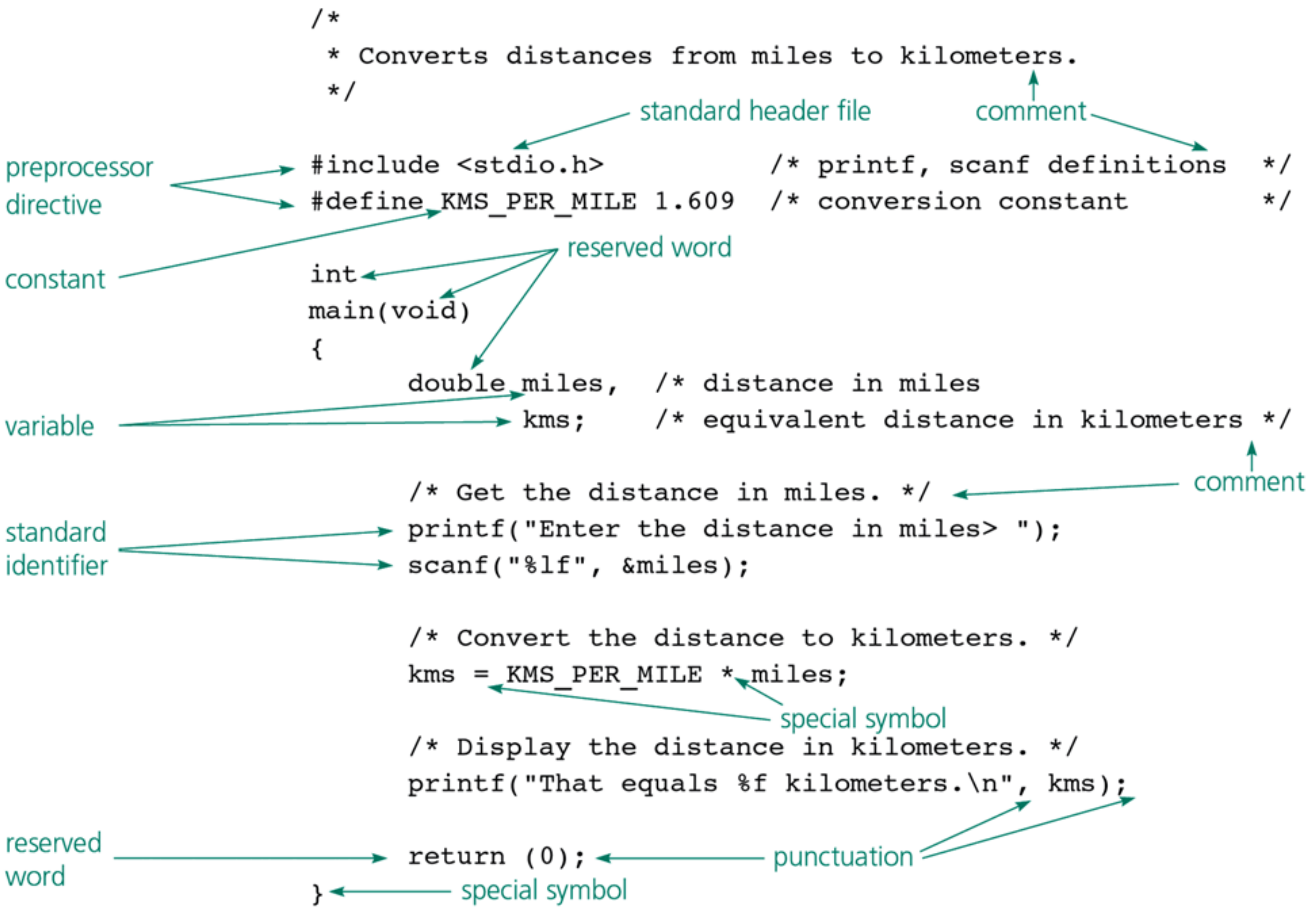
    ```
    int strlen(char string[]);
    ```

    - As we’ve seen, the elements of the array argument are not copied, so the function doesn’t need to know how many elements there are.
  - Array initialization:

    ```
    int vector[] = {1, 2, 3, 4, 5};
    ```

    - In this case, just enough space is allocated to fit all (five) elements of the initializer list

C - Revision



```
/*
 * Converts distances from miles to kilometers.
 */

#include <stdio.h>            /* printf, scanf definitions   */
#define KMS_PER_MILE 1.609    /* conversion constant         */

int
main(void)
{
      double miles,  /* distance in miles
             kms;    /* equivalent distance in kilometers */

      /* Get the distance in miles. */
      printf("Enter the distance in miles> ");
      scanf("%lf", &miles);

      /* Convert the distance to kilometers. */
      kms = KMS_PER_MILE * miles;

      /* Display the distance in kilometers. */
      printf("That equals %f kilometers.\n", kms);

      return (0);
}
```

C Preprocessor

- Preprocessor
  - a system program that modifies the text of a C program before it is compiled
- Preprocessor directives
  - commands that provides instructions to the C preprocessor
    - e.g. #include, #define

C Preprocessor Directives

- #include
  - Notify the preprocessor that it should copy the content of a file at this place
- <stdio.h>
  - Standard header file (provided)
    - Contain function prototypes (declarations) of printf 、 scanf

⇒#include <stdio.h>

- Notify the preprocessor that the file content <stdio.h> will be copied here

C Preprocessor Directives

- #define
  - Using only data values that never change should be given names

    e.g. #define KMS_PER_MILE 1.609

    constant macro constant value

- Constant macro
  - A name that is replaced by a particular constant value
  - Improve the program readability

C Preprocessor Output

Original Source Program

```
#include <stdio.h>
#define KMS_PER_MILE 1.609
int main (void) {
    double miles, kms;
    printf(“Enter the distance in miles> “);
    scanf(“%lf”, &miles);
    kms = KMS_PER_MILE * miles;
    printf(“That equals %f kilomenters.\n”, kms);
    return (0);
}
```

Output After C Preprocessor

```
           :
int printf(const char *format, ...     );  /* printf function prototype */
           :
int scanf(const char *format, ...      );  /* scanf function prototype */
           :
int main (void) {
    double miles, kms;
    printf(“Enter the distance in miles> “);
    scanf(“%lf”, &miles);
    kms = 1.609 * miles;              /* KMS_PER_MILE is replaced by 1.609 */
    printf(“That equals %f kilomenters.\n”, kms);
    return (0);
}
```

C Preprocessor and Compiler

- After C preprocessor finishes, it generates an output that can be viewed as an updated version of the source program.
- The next step is to invoke C compiler.
- The above steps are done automatically after *gcc* utility has been issued.

Algorithm Design - Greedy Algorithms

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Example: Counting Coins

- Suppose you want to count out a certain amount of money, using the fewest possible coins
- A greedy algorithm would do this would be: At each step, take the largest possible coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 coin
    - a $1 coin, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39

C Implementation

```
/* This programs uses the greedy algorithm to give the least
   amount of change
   quarter: 25cent-coin; dime: 10cent-coin; nickel: 5cent-coin;
   penny: 1cent-coin */
# include <stdio.h>

int main(void)
{
   int changeOwed;
   int count = 0;
   int numQ=0, numD=0, numN=0, numP=0;

   printf("How much change is owed (in cents)?\n");
   scanf("%d", &changeOwed);
```

C Implementation (cont’)

```
    int c = changeOwed;// The variable c was only used to shorten my typing
    //Use as many quarters needed
    while(c >= 25){
            count ++;
            numQ++;
            c = c - 25;   }
    //Use as many dimes needed
    while(c >= 10){
            count ++;
            numD++;
            c = c - 10;   }
    //Use as many nickels needed
    while(c >= 5){
            count ++;
            numN++;
            c = c - 5;    }
    //Use as many pennies needed
    while(c >= 1){
            count ++;
            numP++;
            c = c - 1;    }
     }
     printf("Quarters: %d, Dimes: %d, Nickels: %d, Pennies: %d\nNumber of coins
     used= %d\n\n", numQ, numD, numN, numP, count);
}
```

A Failure of The Greedy Algorithm

- In some (fictional) monetary system, “krons” come in 1-kron, 7-kron, and 10-kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10-kron piece
  - Five 1-kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7-kron pieces and one 1-kron piece
  - This only requires three coins
- The greedy algorithm can find a solution, but not an optimal solution