RUTGERS

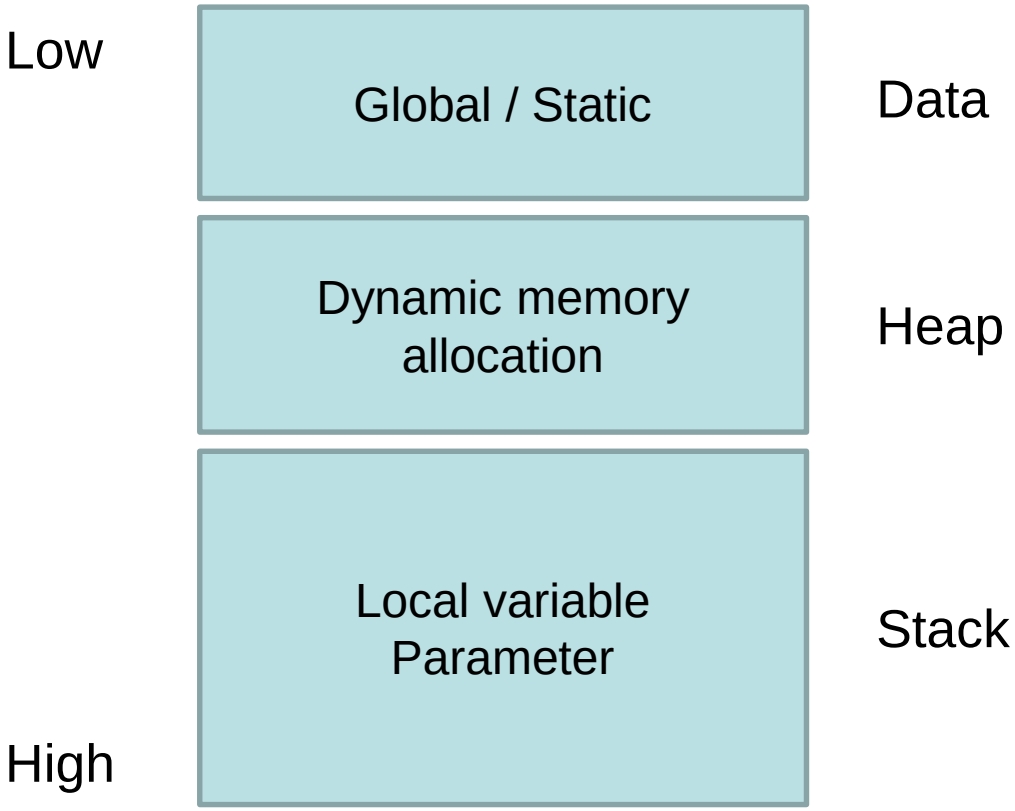
THE STATE UNIVERSITY
OF NEW JERSEY

Recitation 3

Jae Woo Joo

Memory Structure in C

- Stack, Heap, and Data



jaewoo.joo@cs.rutgers.edu

Memory Structure in C

- Data segment
  - Contains any global or static variables
  - The size is determined by the size of the values in the program code, an d does not change at run time
- Heap
  - Grows to larger addresses
  - Managed by programmer (malloc, calloc, realloc, and free)
- Stack
  - Contains local variables and parameters
  - Only created when the function is called

Memory Structure in C

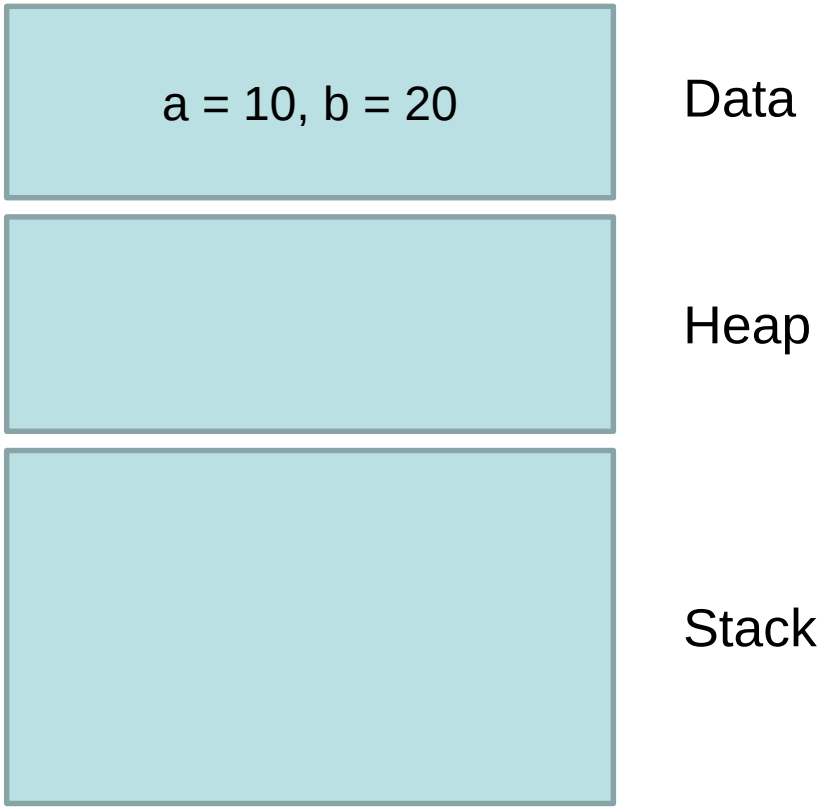
- If you want more information about it:

http://www.geeksforgeeks.org/memory-layout-of-c-program/

Memory flow example

```
1   #include <stdio.h>
2
3   void fct1(int);
4   void fct2(int);
5
6   int a = 10;
7   int b = 20;
8
9   int main() {
10      int m = 123;
11
12      fct1(m);
13      fct2(m);
14
15      return 0;
16  }
17
18  void fc1 (int c) {
19      int d = 30;
20  }
21
22  void fct2 (int e) {
23      int f = 40;
24  }
```

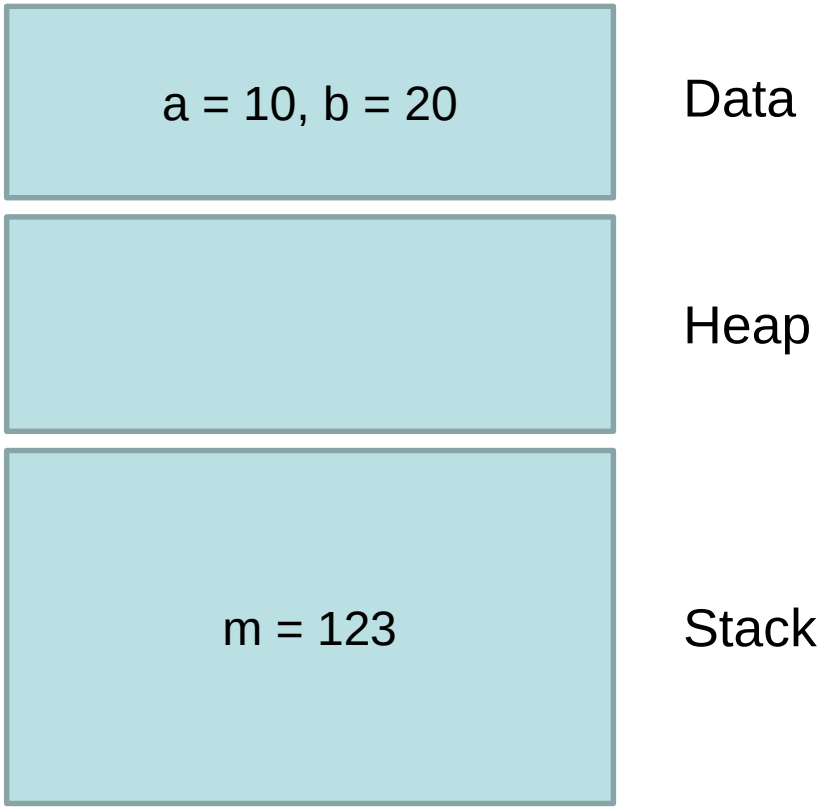
Line 6,7 : global variable



Memory flow example

```
1   #include <stdio.h>
2
3   void fct1(int);
4   void fct2(int);
5
6   int a = 10;
7   int b = 20;
8
9   int main() {
10      int m = 123;
11
12      fct1(m);
13      fct2(m);
14
15      return 0;
16  }
17
18  void fc1 (int c) {
19      int d = 30;
20  }
21
22  void fct2 (int e) {
23      int f = 40;
24  }
```

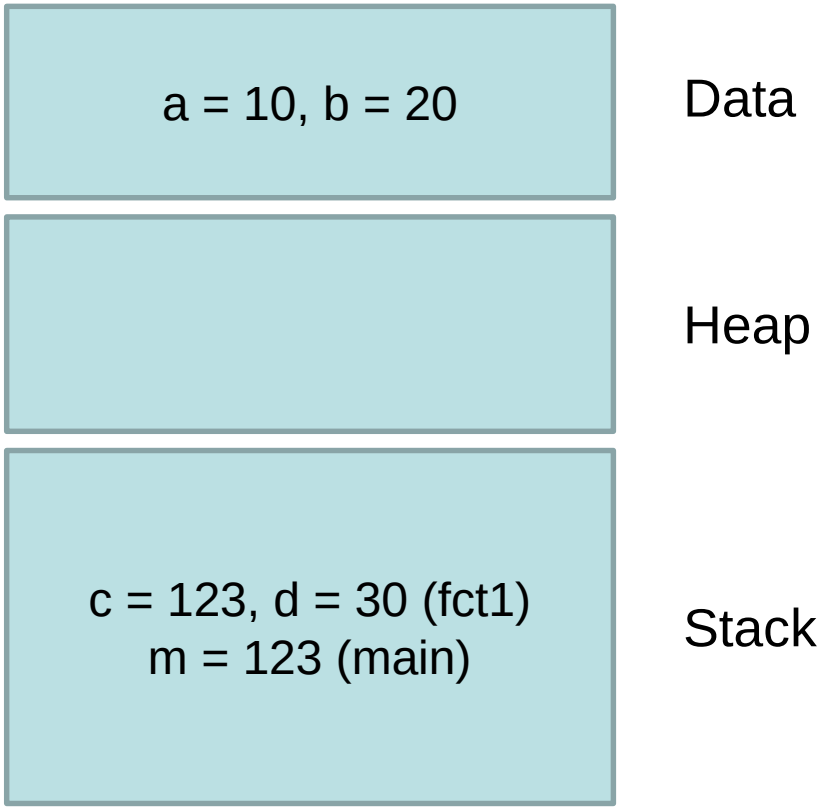
Line 10 : local variable



Memory flow example

```
1   #include <stdio.h>
2
3   void fct1(int);
4   void fct2(int);
5
6   int a = 10;
7   int b = 20;
8
9   int main() {
10      int m = 123;
11
12      fct1(m);
13      fct2(m);
14
15      return 0;
16  }
17
18  void fc1 (int c) {
19      int d = 30;
20  }
21
22  void fct2 (int e) {
23      int f = 40;
24  }
```

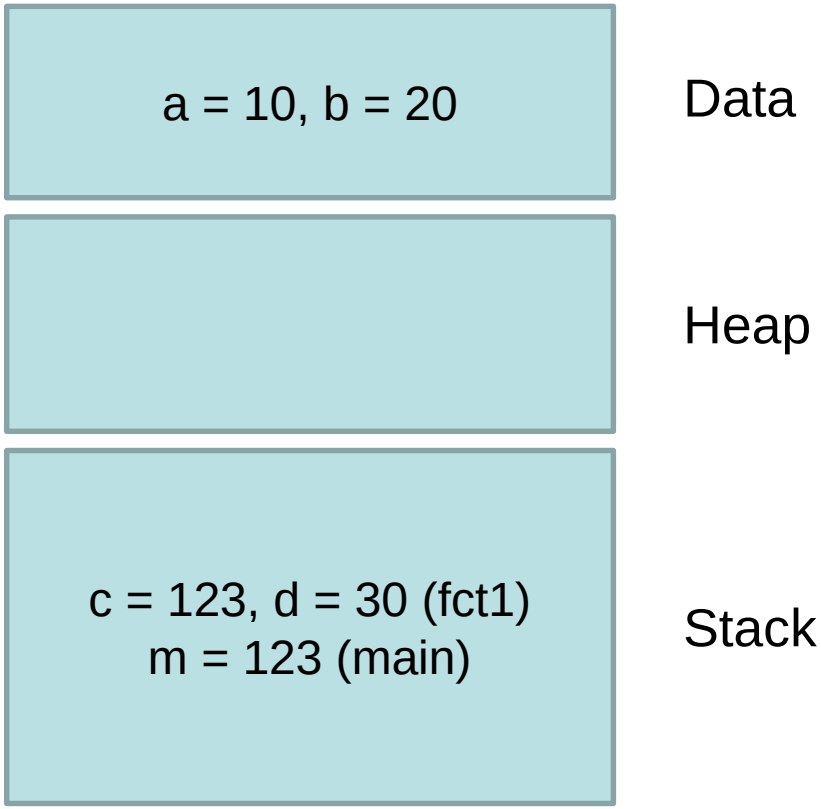
Line 10, 12 : local variable



Memory flow example

```
1   #include <stdio.h>
2
3   void fct1(int);
4   void fct2(int);
5
6   int a = 10;
7   int b = 20;
8
9   int main() {
10      int m = 123;
11
12      fct1(m);
13      fct2(m);
14
15      return 0;
16  }
17
18  void fc1 (int c) {
19      int d = 30;
20  }
21
22  void fct2 (int e) {
23      int f = 40;
24  }
```

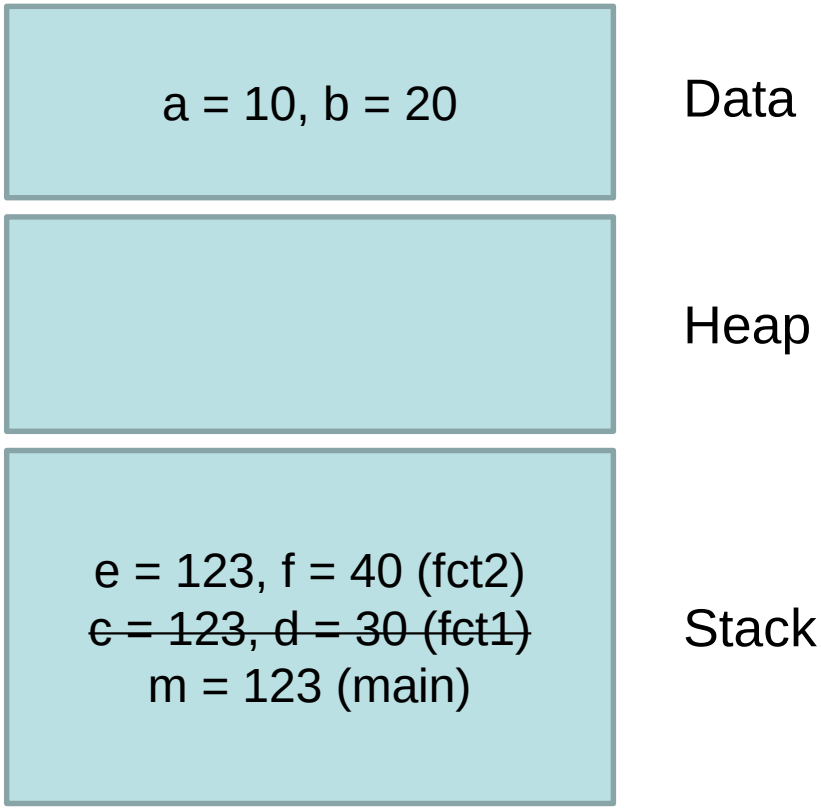
Line 10, 12 : local variable



Memory flow example

```
1   #include <stdio.h>
2
3   void fct1(int);
4   void fct2(int);
5
6   int a = 10;
7   int b = 20;
8
9   int main() {
10      int m = 123;
11
12      fct1(m);
13      fct2(m);
14
15      return 0;
16  }
17
18  void fc1 (int c) {
19      int d = 30;
20  }
21
22  void fct2 (int e) {
23      int f = 40;
24  }
```

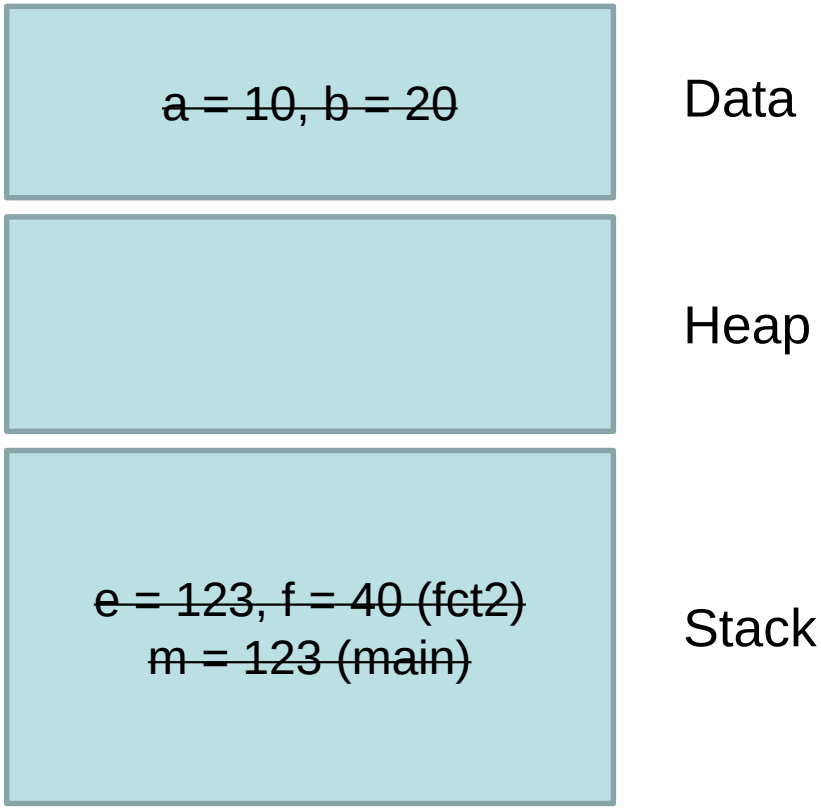
Line 10, 13 : local variable



Memory flow example

```c
#include <stdio.h>

void fct1(int);
void fct2(int);

int a = 10;
int b = 20;

int main() {
    int m = 123;

    fct1(m);
    fct2(m);

    return 0;
}

void fc1 (int c) {
    int d = 30;
}

void fct2 (int e) {
    int f = 40;
}
```

End function main, fct1, fct2



Memory allocation

- The memory allocated in stack and data segment will be deter mined when it is compiling

- How much memory do we need?

```
void function (int a) {
    int b;
    int c[2];
}
```

- a-4, b-4, c-8 -> total-16 bytes

Memory allocation

- Then how much memory do we need?

```
void function (void) {
    int i = 10;
    int array[i];
}
```

- The compiler cannot calculate!
- Why?

Dynamic Memory Allocation

- What is dynamic allocation?
  - Allocating memory inside the **heap**
  - Malloc(), calloc(), realloc(), free()
  - These functions re containted in <stdlib.h> header file
  - Allocating memory inside **stack** and **data segment** is called **static alloc ation**

Dynamic Memory Allocation

- Malloc ()
  - Reserves the number of bytes requested by the argument passed to fun ction
  - Returns the address of the first reserved location
  - **NULL** if there is insufficient memory

- Calloc ()
  - Reserves space for an array of n elements of specified size
  - Returns the address of the first reserved location
  - **NULL** if there is insufficient memory

Dynamic Memory Allocation

- Realloc ()
  - Changes the size of previously allocated memory to new size
- Free ()
  - Releases a block of bytes previously reserved
  - The address of the first reserved location is passed as an argument to the function

Dynamic Memory Allocation

- The **malloc ()** and **calloc ()** functions can frequently be used interchangeably
- We use **malloc()** because it is the more general purpose of the two functions
- Example)
  - malloc (10 * sizeof(char))
  - calloc (10, sizeof(char))
  - Both requests enough memory to store 10 characters

Dynamic Memory Allocation

- To provide access to locations, malloc() returns the address of the first location that is reserved
- Address must be assigned to a pointer
- Especially useful for creating arrays

Dynamic Memory Allocation

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

int main() {
    int numgrades, i;
    int *grades;

    printf("\n Enter the number of grades to be processed: ");
    scanf("%d", &numgrades);

    /* here is where the request for memory is made */
    grades = (int *) malloc(numgrades * sizeof(int));

    /* here we check that the allocation was satisfied */
    if (grades == (int *) NULL) {
        printf("\n Failed to allocate grades array \n");
        exit(1);
    }

    for (i = 0; i < numgrades; i++) {
        printf("Enter a grade: ");
        scanf("%d", &grade[i]);
    }

    printf("\n An array was created for %d integers", numgrades);
    printf("\n The values stored in the array are: \n");

    for (i = 0; i <numgrades; i++)
        printf("%d \n", grades[i]);

    free(grades);

    return 0;
}
```

Restore the allocated block of storage back to the oper ating system at the end

Dynamic Memory Allocation

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

int main() {
    int numgrades, i;
    int *grades;

    printf("\n Enter the number of grades to be processed: ");
    scanf("%d", &numgrades);

    /* here is where the request for memory is made */
    grades = (int *) malloc(numgrades * sizeof(int));

    /* here we check that the allocation was satisfied */
    if (grades == (int *) NULL) {
        printf("\n Failed to allocate grades array \n");
        exit(1);
    }

    for (i = 0; i < numgrades;
        printf("Enter a grade:
        scanf("%d", &grade[i]);
    }

    printf("\n An array was cre
    printf("\n The values store

    for (i = 0; i <numgrades; i
        printf("%d \n", grades[

    free(grades);

    return 0;
}
```

```
Enter the number of grades to be processed:
4
Enter a grade: 85
Enter a grade: 96
Enter a grade: 77
Enter a grade: 92

An array was created for 4 integers
The values stored in the array are:
```

Structure

- Complex data type declaration that defines a physically group ed list of variables under one name
- Structure's **form** consists of symbolic names, data types, and arrangement of individual data fields in the record
- Structure's **contents** consist of the actual data stored in the s ymbolic names

Structure

- Structure definition in C

```
struct {
        int month;
        int day;
        int year;
} birth
```

  - Reserves storage for the individual data items listed in the structure
  - Three data items are the members of the structure

Structure

- Multiple variables can be defined in one statement
  - struct {int month; int day; int year; } birth, current, …;
- Common to list the form of the structure with no following variable names
  - The list of structure members must be preceded by a user-selected structure type name

```
struct Date {
        int month;
        int day;
        int year;
};
```

struct Date birth, current;

Structure

Program 12.2

```
 1  #include <stdio.h>
 2  struct Date
 3  {
 4    int month;
 5    int day;
 6    int year;
 7  };
 8
 9  int main()
10  {
11    struct Date birth;
12
13    birth.month = 12;
14    birth.day = 28;
15    birth.year = 1987;
16    printf("My birth date is %d/%d/%d\n",
17    birth.month,birth.day,birth.year % 100);
18
19    return 0;
20  }
```

By convention the first letter of user-selected structure type names is uppercase

Typedef Statement

- A commonly used programming technique when dealing with structure declarations

```
struct Date {
    int month;
    int day;
    int year;
};

typedef struct Date DATE;

struct Date a, b, c;

DATE a, b, c;
```


These two statements are same

Passing and Returning Structures

- Individual structure members may be passed to a function
  - Ex) display (emp.idNum);
- On most compilers, complete copies of all members of a struc ture can be passed to a function by including the name of the structure as an argument
  - Ex) calcNet (emp);

Passing and Returning Structures

Program 12.4

```
 1  #include <stdio.h>
 2  struct Employee /* declare a global structure type */
 3  {
 4    int idNum;
 5    double payRate;
 6    double hours;
 7  };
 8
 9  double calcNet(struct Employee); /* function prototype */
10
11  int main()
12  {
13    struct Employee emp = {6787, 8.93, 40.5};
14    double netPay;
15
16    netPay = calcNet(emp); /* pass copies of the values in emp */
17    printf("The net pay of employee %d is $%6.2f\n",
18                                          emp.idNum,netPay);
19
20    return 0;
21  }
22
23  double calcNet(struct Employee temp)
24  /* temp is of data type struct Employee */
25  {
26    return(temp.payRate * temp.hours);
27  }
```

Passing and Returning Structures

Program 12.4

```
 1  #include <stdio.h>
 2  struct Employee /* declare a global structure type */
 3  {
 4    int idNum;
 5    double payRate;
 6    double hours;
 7  };
 8
 9  double calcNet(struct Employee); /* function prototype */
10
11  int main()
12  {
13    struct Employee emp = {6787, 8.93, 40.5};
14    double netPay;
15
16    netPay = calcNet(emp); /* pass copies of the values in emp */
17    printf("The net pay of employee %d is $%6.2f\n",
18                                          emp.idNum,netPay);
19
20    return 0;
21  }
22
23  double calcNet(struct Employee temp)
24  /* temp is of data type struct Employee */
25  {
26    return(temp.payRate * temp.hours);
27  }
```

The net pay of employee 6787 is $ 361.66

Passing and Returning Structures

- A structure can be passed by reference
  - calcNet (&emp);
  - double calcNet (struct Employee *pt);
  - (*pt).idNum or
    *pt -> idNum

Passing and Returning Structures

Program 12.5

```
 1  #include <stdio.h>
 2  struct Employee /* declare a global structure type */
 3  {
 4    int idNum;
 5    double payRate;
 6    double hours;
 7  };
 8
 9  double calcNet(struct Employee *); /* function prototype */
10
11  int main()
12  {
13    struct Employee emp = {6787, 8.93, 40.5};
14    double netPay;
15
16    netPay = calcNet(&emp); /* pass an address*/
17    printf("The net pay for employee %d is $%6.2f\n",
18                                      emp.idNum, netPay);
19
20    return 0;
21  }
22
23  double calcNet(struct Employee *pt) /* pt is a pointer to a */
24  {                              /* structure of Employee type */
25
26    return(pt->payRate * pt->hours);
27  }
```

Passing and Returning Structures

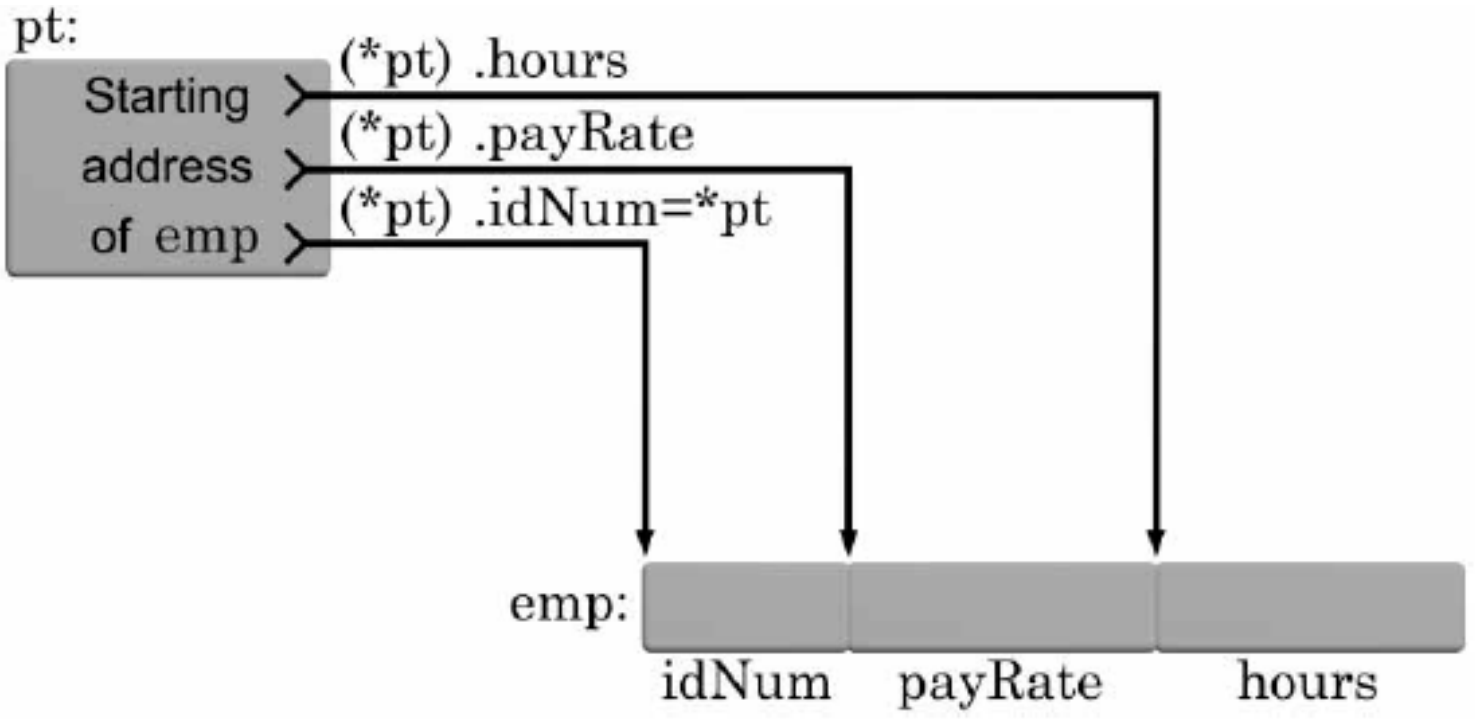


Figure 12.5 A pointer can be used to access structure members

Union

- A data type that reserves the same area in memory for two or more variables

```
union {
        char key;
        int num;
        double price;
} val;
```

  - Each of these types, but only one at a time, can be assigned to the union variable
  - Reserves sufficient memory locations to its largest member's data type

Structure vs Union

Structure	Union
The size of structure is greater or equal to the sum of sizes of its members	The size of union is equal to the size of largest member
The memory for each member will start at different offset values	The address is same for all the members of a union
Altering the value of a member will not affect other members of the structure	Altering the value of any member will alter other member values
Individual member can be accessed at a time	Only one member can be accessed at a time

Number System

Decimal	Binary	Octal	Hexa-decimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7

Decimal	Binary	Octal	Hexa-decimal
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Converting Hex to Binary

- Each hex digit can be represented by 4 binary digits
  - Why?
- Example
  - 0x2A8C (hex) = 0b0010101010001100 (binary)
- So to convert hex to binary, just convert each digit and concatenate

Converting Binary to Hex

- Do it reverse
  - Group each set of 4 digits and change to corresponding digit in hex
  - Go from right to left

- Example
  - 0b1011011111001100 = 0xB79C

Decimal and Binary fractions

- In decimal, digits to the right of radix point have value $1/10^i$ for each digit in the i^{th} place
- Similarly, in binary, digits to the right of radix point have value $1/2^i$ for each i^{th} place
- Example
  - 0.625 (decimal) = 0.101 (binary)
- How to convert?

Decimal and Binary fractions

- Begin with the decimal fraction and multiply by 2. Keep the first binary digit
  - 0.625 * 2 = **1**.25
  - So far 0.625 =0.1?? (binary)
- Follow the same rule by multiplying 2 with the remaining digit
  - 0.25*2 = **0**.50
  - So far 0.625 = 0.10? (binary)
- Continue this process until we get a zero as our decimal part
  - 0.50*2 = **1**.00
  - So far 0.625 = 0.101 (binary)
- Because we have 0 as a fractional part we can end this process

Q&A

- Any questions?