Programming for Engineers

Lecture 11: Bit Operations/File Processing/Dynamic Memory Allocation

Course ID: EE057IU

Outline

- Bitwise Operations
- File Processing
- Dynamic Memory Allocation

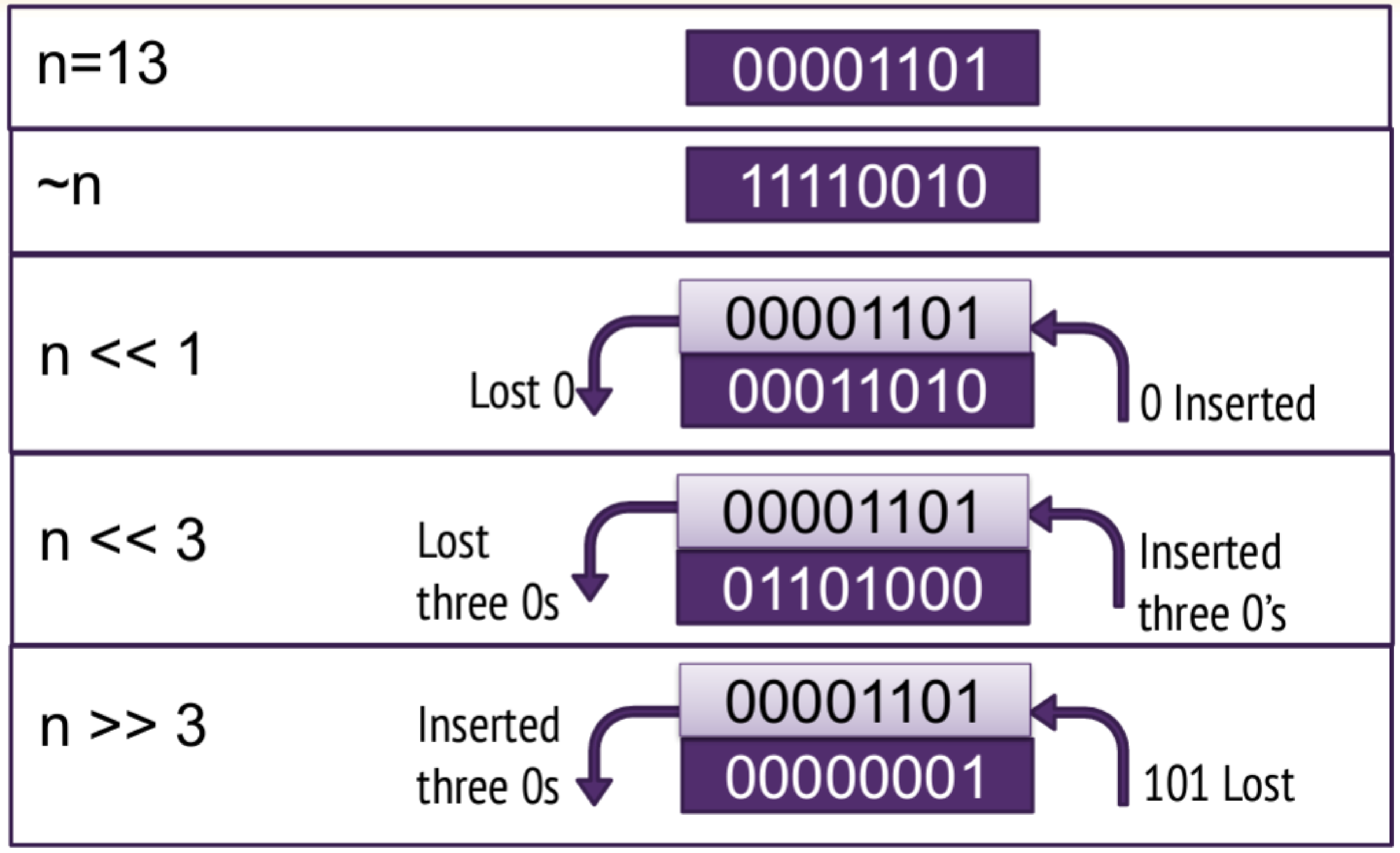
Bitwise Operations

- Computers represent all data internally as sequences of bits.
- Each bit can assume the value 0 or the value 1.
- The bitwise operators are used to manipulate the bits of integral operands both signed and unsigned.

Bitwise Operator

Operator		Description
&	bitwise AND	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are *both* 1.
|	bitwise inclusive OR	Compares its two operands bit by bit. The bits in the result are set to 1 if *at least one* of the corresponding bits in the two operands is 1.
^	bitwise exclusive OR (also known as bitwise XOR)	Compares its two operands bit by bit. The bits in the result are set to 1 if the corresponding bits in the two operands are different.
<<	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand; fill from the right with 0 bits.
>>	right shift	Shifts the bits of the first operand right by the number of bits specified by the second operand; the method of filling from the left is machine-dependent when the left operand is negative.
~	complement	All 0 bits are set to 1, and all 1 bits are set to 0. This is often called toggling the bits.

Bitwise Operation Example



Bitwise Operation Example

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
0	1	0
1	0	0
1	1	1

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
0	1	1
1	0	1
1	1	1

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
0	1	1
1	0	1
1	1	0

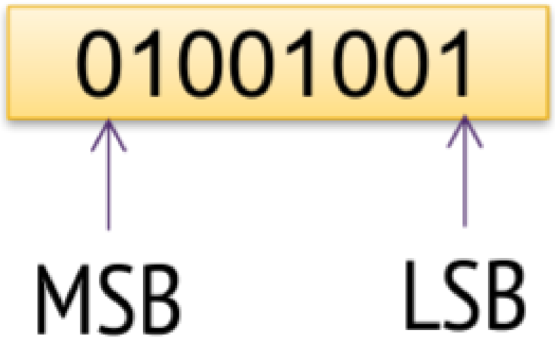
n	00001101
m	01010101
n&m	00000101

n	00001101
m	01010101
n|m	01011101

n	00001101
m	01010101
n^m	01011000

Bitwise Order

- Most Significant Bit (MSB)
- Least Significant Bit (LSB)



Field Extraction using Mask

How to isolate (extract) specific bits from a string of data while discarding the rest?

- ANDing a bit with 0 produces 0.
- ANDing a bit with 1 produces the original bit.

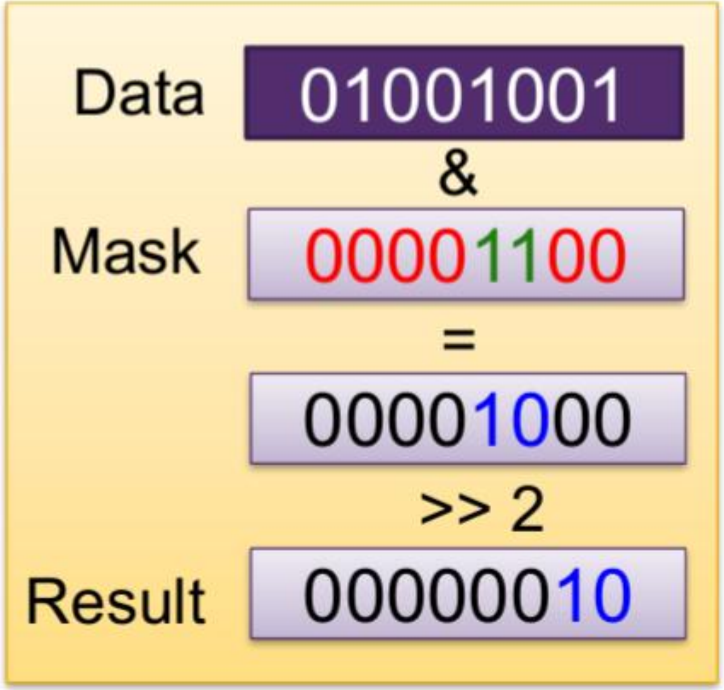
Data	01001101
Only rightmost two bits needed	01

"Mask"

A stencil used in spray painting covers up the parts you don't want to paint and exposes the parts you do

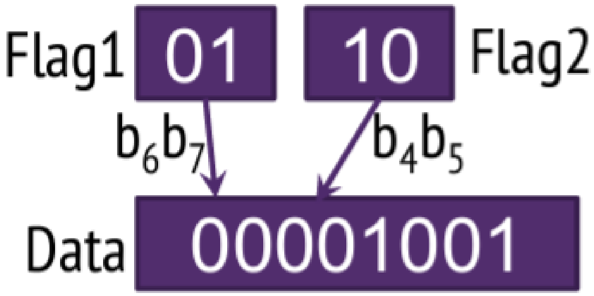
Data	01001101
	&
Mask	00000011
	=
Result	00000001

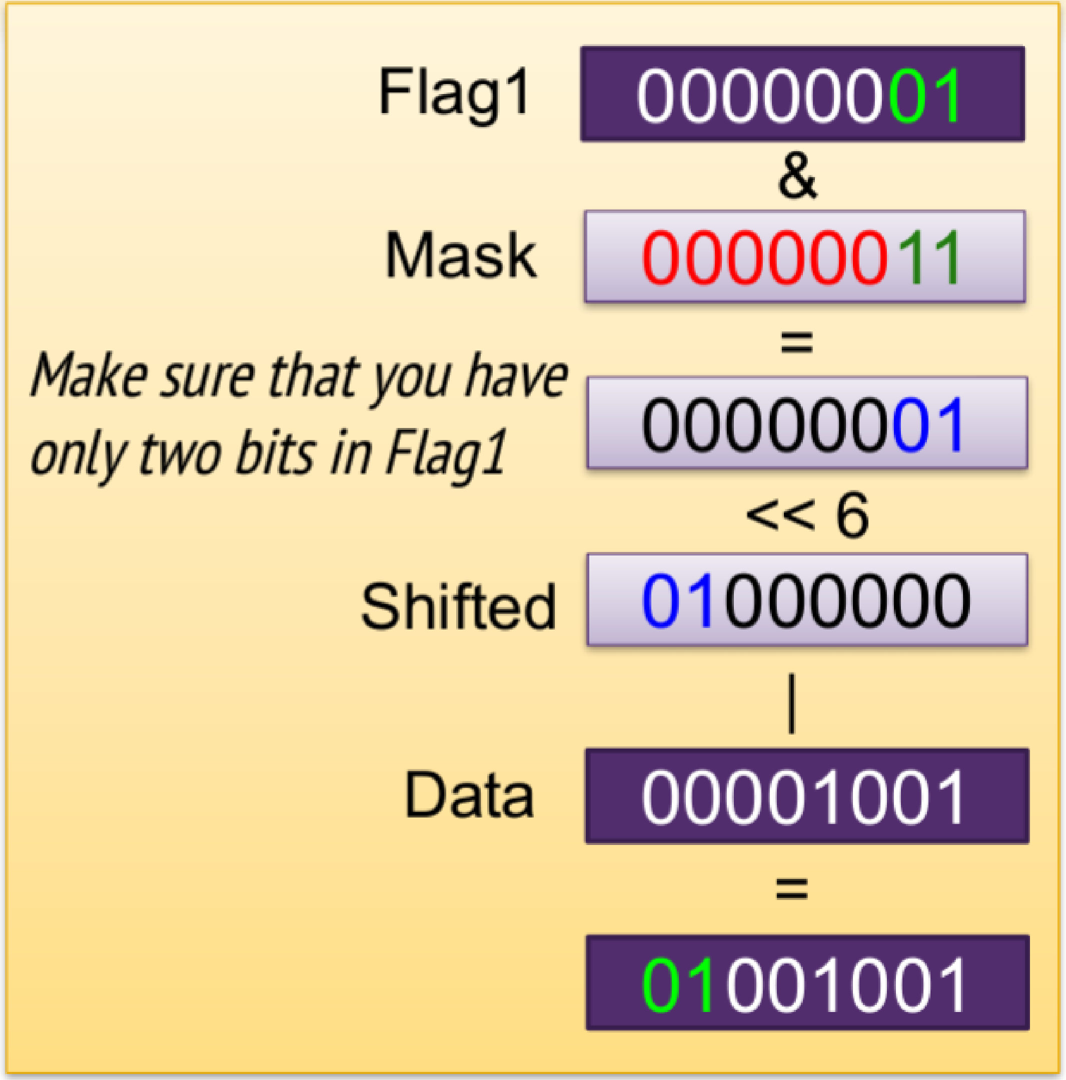
Field Extraction using Mask and Shift



Field Insertion

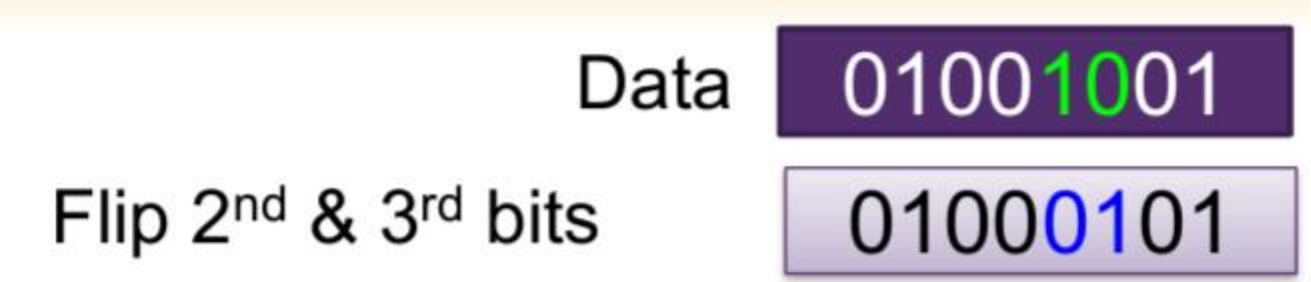
You'd like to insert bits of Flag 1 into your data at b_6 and b_7.

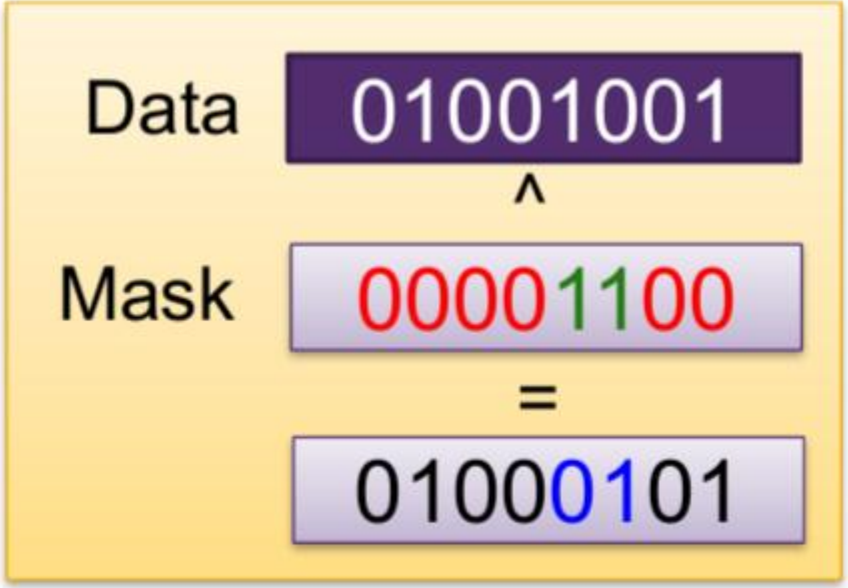




Flip bits

Using OR operation to flip bits





Display Unsigned Interger’s Bits

```c
#include <stdio.h>

void displayBits(unsigned int value);

int main(void) {
    unsigned int x = 0;

    printf("%s", "Enter a nonnegative int: ");
    scanf("%u", &x);

    displayBits(x); // Call the function
}
```

Output

```
Enter a nonnegative int: 7
         7 = 00000000 00000000 00000000 00000111
```

```c
void displayBits(unsigned int value) {
    // 1. Create a mask for the High Order bit (1000...)
    unsigned int displayMask = 1 << 31;

    printf("%10u = ", value);

    // 2. Loop through all 32 bits
    for (unsigned int c = 1; c <= 32; ++c) {

        // Check if the leading bit is 1 or 0
        putchar(value & displayMask ? '1' : '0');

        // Shift value left to bring the next bit forward
        value <<= 1;

        // Format: Add a space every 8 bits
        if (c % 8 == 0) {
            putchar(' ');
        }
    }
    putchar('\n');
}
```

Bitwise Operations Example

```c
// fig10_05.c
// Using the bitwise AND, bitwise inclusive OR, bitwise
// exclusive OR and bitwise complement operators
#include <stdio.h>

void displayBits(unsigned int value); // prototype

int main(void) {
   // demonstrate bitwise AND (&)
   unsigned int number1 = 65535;
   unsigned int mask = 1;
   puts("The result of combining the following");
   displayBits(number1);
   displayBits(mask);
   puts("using the bitwise AND operator & is");
   displayBits(number1 & mask);

   // demonstrate bitwise inclusive OR (|)
   number1 = 15;
   unsigned int setBits = 241;
   puts("\nThe result of combining the following");
   displayBits(number1);
   displayBits(setBits);
   puts("using the bitwise inclusive OR operator | is");
   displayBits(number1 | setBits);

   // demonstrate bitwise exclusive OR (^)
   number1 = 139;
   unsigned int number2 = 199;
   puts("\nThe result of combining the following");
   displayBits(number1);
   displayBits(number2);
   puts("using the bitwise exclusive OR operator ^ is");
   displayBits(number1 ^ number2);

   // demonstrate bitwise complement (~)
   number1 = 21845;
   puts("\nThe one);
   displayBits(number1);
   puts("is");
   displayBits(~number1);
}

// display bits of an unsigned int value
void displayBits(unsigned int value) {
   // declare displayMask and left shift 31 bits
   unsigned int displayMask = 1 << 31;

   printf("%10u = ", value);

   // loop through bits
   for (unsigned int c = 1; c <= 32; ++c) {
      putchar(value & displayMask ?   : );
      value <<= 1; // shift value left by 1

      if (c % 8 == 0) { // output a space after 8 bits
         putchar();
      }
   }

   putchar();
}
```

Bitwise Operations Example

```
The result of combining the following
     65535 = 00000000 00000000 11111111 11111111
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

The result of combining the following
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       255 = 00000000 00000000 00000000 11111111

The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
        76 = 00000000 00000000 00000000 01001100

The one
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

Bitwise Left- and Right-shift Operators (1)

```c
// fig10_06.c
// Using the bitwise shift operators
#include <stdio.h>

void displayBits(unsigned int value); // prototype

int main(void) {
   unsigned int number1 = 960; // initialize number1

   // demonstrate bitwise left shift
   puts("\nThe result of left shifting");
   displayBits(number1);
   puts("8 bit positions using the left shift operator << is");
   displayBits(number1 << 8);

   // demonstrate bitwise right shift
   puts("\nThe result of right shifting");
   displayBits(number1);
   puts("8 bit positions using the right shift operator >> is");
   displayBits(number1 >> 8);
}
```

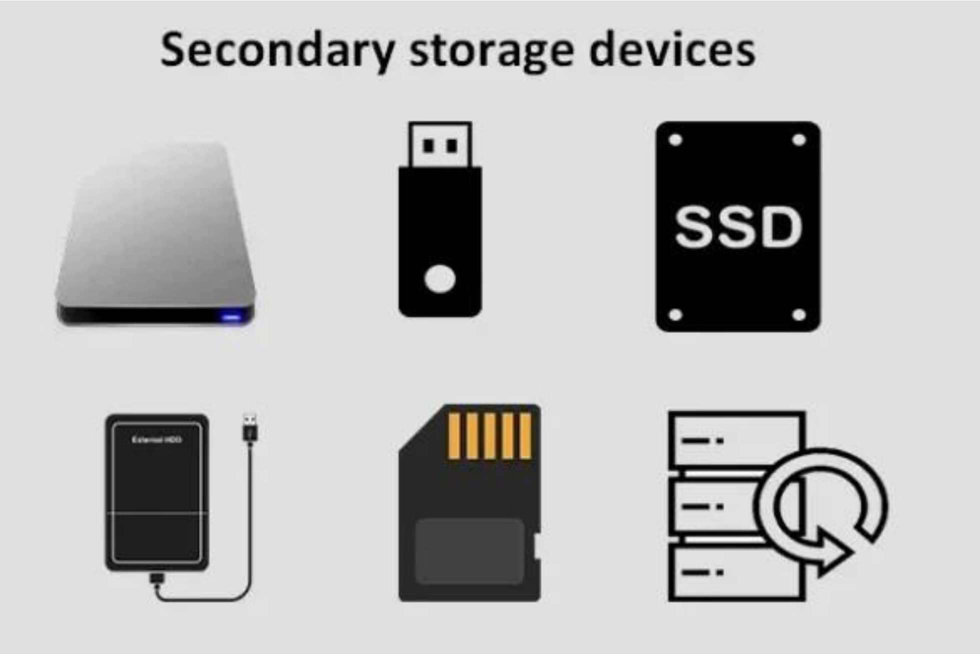
Bitwise Left- and Right-shift Operators (1)

```c
// display bits of an unsigned int value
void displayBits(unsigned int value) {
   // declare displayMask and left shift 31 bits
   unsigned int displayMask = 1 << 31;

   printf("%10u = ", value);

   // loop through bits
   for (unsigned int c = 1; c <= 32; ++c) {
      putchar(value & displayMask ?  : );
      value <<= 1; // shift value left by 1

      if (c % 8 == 0) { // output a space after 8 bits
         putchar();
      }
   }

   putchar();
}
```

```
The result of left shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
    245760 = 00000000 00000011 11000000 00000000

The result of right shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
         3 = 00000000 00000000 00000000 00000011
```

File Processing in C

- Data in variables is $temporary$
- It's lost when a program terminates
- Files are used for permanent retention of data
- Computers store files on secondary storage devices



Files and Streams

- When you open a file, C associates a **stream** with it. When program execution begins, C opens three streams automatically:
  - The **standard input** stream receives input from the keyboard.
  - The **standard output** stream displays output on the screen.
  - The **standard error** stream displays error messages on the screen.

- Text file vs Binary files
  - Text file is a term used for a file that is essentially a sequence of character codes.
  - Binary file is a term used for a file in which most bytes are not intended to be interpreted as character codes.

Steps in processing a file

- Create the stream via a pointer variable using the FILE structure:

  FILE *p;

- Open the file, associating the stream name with the file name.
- Read or write the data.
- Close the file.

Opening Binary Files

Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Open or create a file for writing at the end of a file—this is for write operations that append data to a file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for reading and writing. If the file already exists, discard the current contents.
a+	Open or create a file for reading and updating where all writing is done at the end of the file—that is, write operations append data to the file.
rb	Open an existing binary file for reading.
wb	Create a binary file for writing. If the file already exists, discard the current contents.
ab	Open or create a binary file for writing at the end of the file (appending).
rb+	Open an existing binary file for update (reading and writing).
wb+	Create a binary file for update. If the file already exists, discard the current contents.
ab+	Open or create a binary file for update. Writing is done at the end of the file.

Opening File using fopen() and Create a File for writing

```c
// fig11_01.c
// Creating a sequential file
#include <stdio.h>
int main(void) {
    // cfPtr = clients.txt file pointer
    FILE *cfPtr = NULL;
    // fopen opens the file. Exit the program if unable to create the file
    if ((cfPtr = fopen("clients.txt", "w")) == NULL) {
        puts("File could not be opened");
    }
    else {
        puts("Enter the account, name, and balance.");
        puts("Enter EOF to end input.");
        printf("%s", "? ");

        int account = 0; // account number
        char name[30] = ""; // account name
        double balance = 0.0; // account balance

        scanf("%d%29s%lf", &account, name, &balance);

        // write account, name and balance into file with fprintf
        // Checks if the user has signaled the "End of File"
        // by pressing Ctrl+Z on Windows or Ctrl+D on Mac/Linux
        while (!feof(stdin)) {
            fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
            printf("%s", "? ");
            scanf("%d%29s%lf", &account, name, &balance);
        }
        fclose(cfPtr); // fclose closes file
    }
}
```

- FILE *fopen(const char *filename, const char *mode);

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Functions to read and write data to file

- Function fgets
  - Reads one line from a file.
  - char *fgets(char *str, int n, FILE *stream)
  - Like printf
  -
- Function fputs
  - Writes one line to a file.
  - int fputs(const char *str, FILE *stream)
  - Like scanf

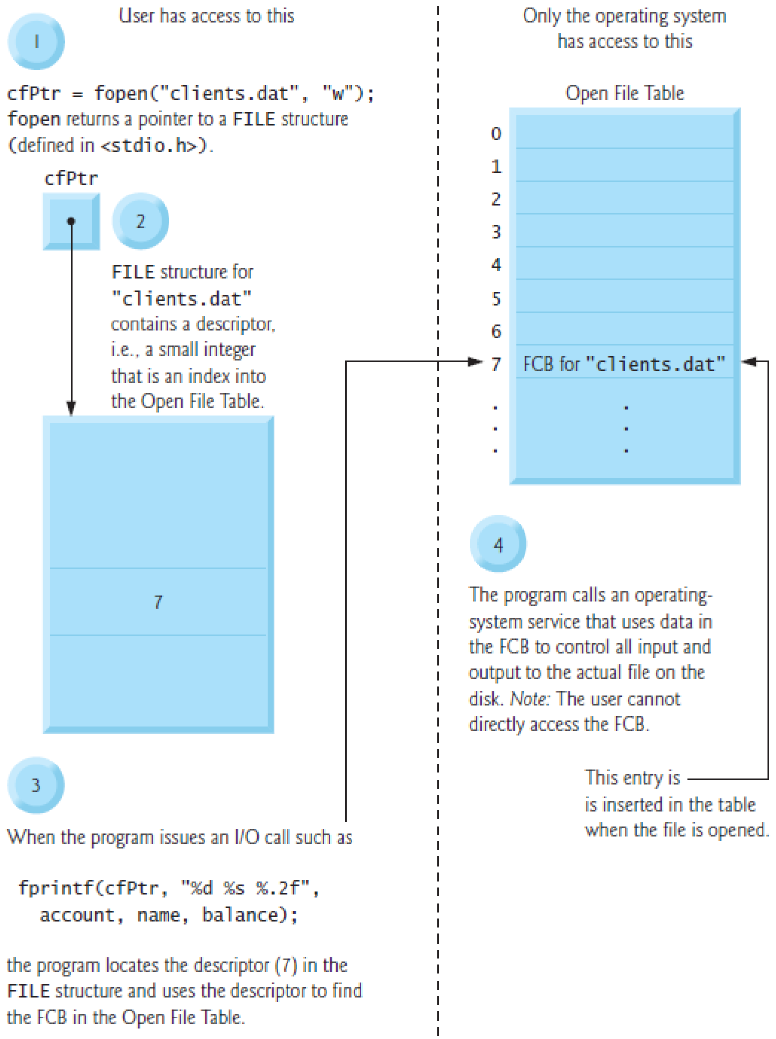
Close the File: fclose()

- int fclose(FILE * stream)
- Returns 0 if successfully closed
- *If function $fclose$ is not called explicitly, the operating system normally will close the file when program execution terminates.*

Reading Data from a Sequential-Access File

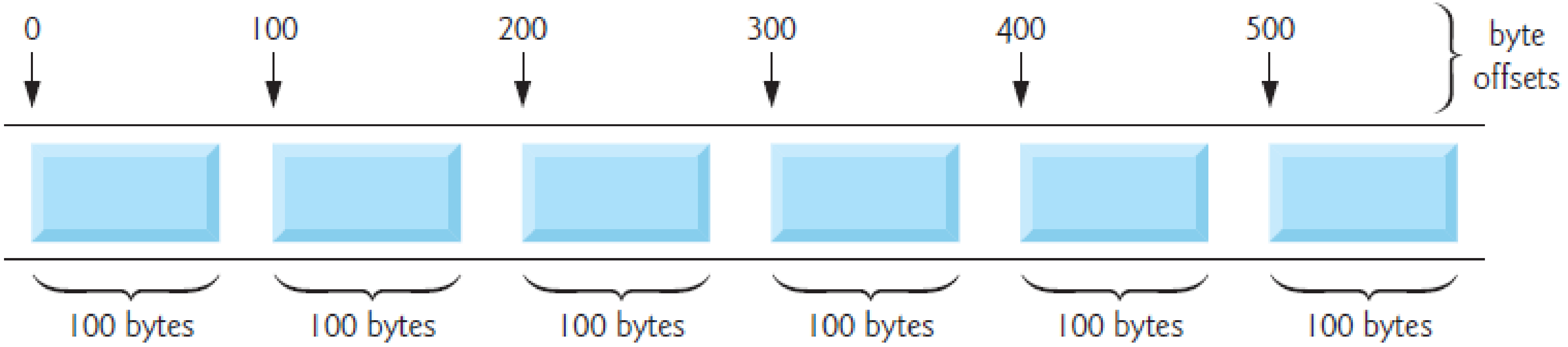
```c
// fig11_02.c
// Reading and printing a sequential file
#include <stdio.h>
int main(void) {
    FILE *cfPtr = NULL; // cfPtr = clients.txt file pointer
    // fopen opens file; exits program if file cannot be opened
    // Note the "r" for READ mode
    if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
        puts("File could not be opened");
    }
    else {
        // Create variables to hold the data coming FROM the file
        int account = 0;
        char name[30] = "";
        double balance = 0.0;

        printf("%-10s%-13s%s\n", "Account", "Name", "Balance");

        // Read the first item from the file
        fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);

        // Loop while not end of file
        while (!feof(cfPtr)) {
            printf("%-10d%-13s%7.2f\n", account, name, balance);
            // Read the next item
            fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
        }
        fclose(cfPtr); // fclose closes the file
    }
}
```

FILE Pointer



Random Access File

- Individual records of a random-access file are normally fixed in length and may be accessed directly (and thus quickly) without searching through other records.
- Random-access files are appropriate for
  - airline reservation systems, banking systems, point-of-sale systems, and other kinds of transaction-processing systems that require rapid access to specific data.



$fwrite()$

- Example use

  $fprintf(fPtr, \text{"\%d"}, number);$

  could print a single digit or as many as 11 digits (10 digits plus a sign, each of which requires 1 byte of storage)

- For a four-byte integer, we can use

  $fwrite(\&number, sizeof(int), 1, fPtr);$

  which always writes four bytes on a system with fourbyte integers from a variable number to the file represented by fPtr. 1 denotes one integer will be written.

$fread()$

- Function fread reads a specified number of bytes from a file into memory.
- For example,

$$fread(\&client, sizeof(struct\ clientData), 1, cfPtr);$$

reads the number of bytes determined by sizeof(struct clientData) from the file referenced by cfPtr, stores the data in client and returns the number of bytes read.

Moving to a location – fseek()

- fseek

$$int\ fseek(FILE\ *stream, long\ int\ offset, int\ whence);$$

  - offsetis the number of bytes to seek from
  - whencein the file pointed to by stream—a positive offset seeks forward and a negative one seeks backward.
- Argument whence is one of the values
  - SEEK_SET: Value 0, beginning of file.
  - SEEK_CUR: Value 1, current position.
  - SEEK_END: Value 2, end of file.

Random Access File Code

```c
// fig11_04.c
// Creating a random-access file sequentially
#include <stdio.h>

// clientData structure definition
struct clientData {
   int account;
   char lastName[15];
   char firstName[10];
   double balance;
};

int main(void) {
   FILE *cfPtr = NULL; // accounts.dat file pointer

   // fopen opens the file; exits if file cannot be opened
   if ((cfPtr = fopen("accounts.dat", "wb")) == NULL) {
      puts("File could not be opened.");
   }
   else {
      // create clientData with default information
      struct clientData blankClient = {0, "", "", 0.0};

      // output 100 blank records to file
      for (int i = 1; i <= 100; ++i) {
         fwrite(&blankClient, sizeof(struct clientData), 1, cfPtr);
      }

      fclose (cfPtr); // fclose closes the file
   }
}
```

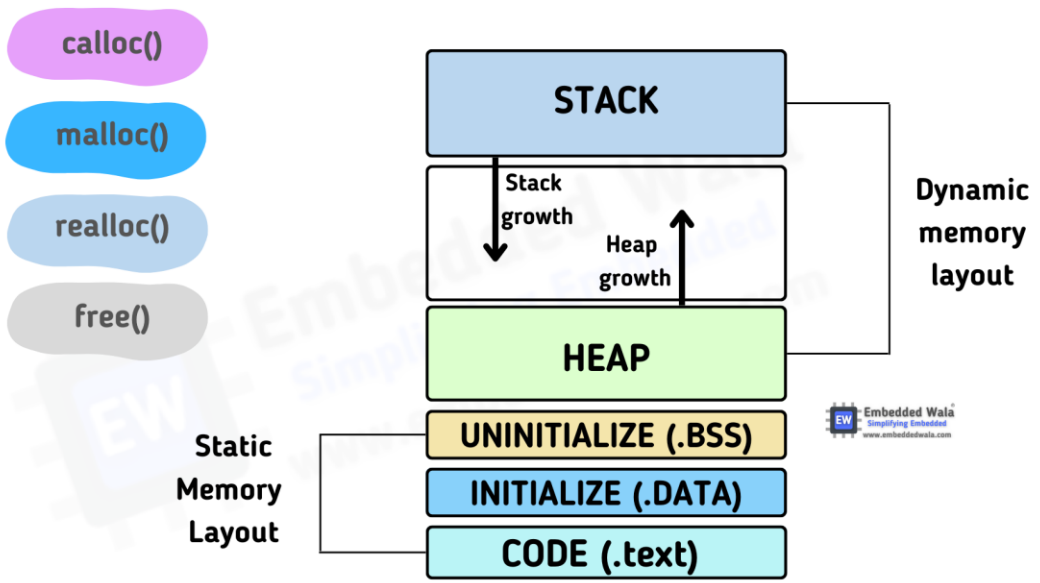
Section Break

Memory Allocation

- Arrays are better than linked lists for rapid sorting, searching and data access
- Arrays are normally **static data structures** that cannot be resized

- For all data, memory must be allocated
  - Allocated = memory space reserved

- This raises two question
  - When do we know the size to allocate?
  - When do we allocate?



How much memory to allocate?

- Sometimes it is obvious:

```
char   c;
int    array[10];
```

One byte

10 * sizeof(int) (= 40 on CLEAR)

- Sometimes it is not

```
char *c;
int  *array;
```

Is this going to point to one character or a string?

How big will this array be?

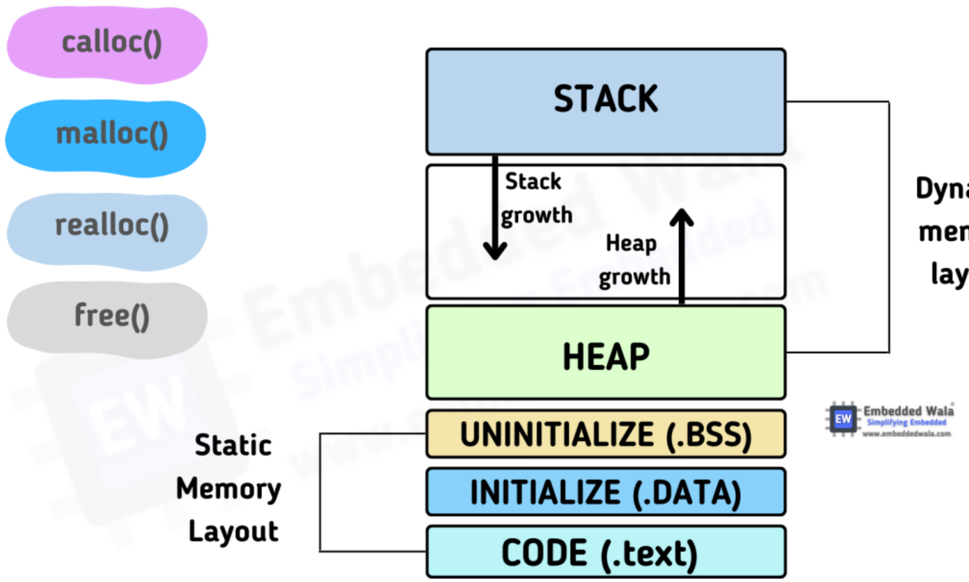
- Question:
  - Will they point to already allocated memory?
  - Will the new memory need to be allocated?

Dynamic Memory Allocation

- Creating and maintaining dynamic data structures requires dynamic memory allocation
  - the ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed
- Functions malloc and free, and operator sizeof, are essential to dynamic memory allocation.

Memory use

- heap
  - region of memory in which function malloc dynamically allocates blocks of storage
- stack
  - region of memory in which function data areas are allocated and reclaimed



malloc()

- void * $malloc$ (size_t size)
  - Input: pass to the function $malloc$ number of byte to allocate
  - Output: a poiter of type void * (pointer to void) to the allocated memory

- Example

```
newPtr = malloc(sizeof(int));
```

- The allocated memory is not initialize
- If no memory is available, malloc returns NULL

free()

- If we no longer need block of dynamic allocated memory
  - Deallocate memory by function $free()$
- To free memory dynamically allocated by the preceding $malloc()$

  $free(newPtr);$
- C also provides functions calloc and realloc for creating and modifying *dynamic arrays*

calloc() and realloc()

- calloc()
  - Allocate memory and initializes array's elements to zeros (0)
  - void *calloc(size_t nmemb, size_t size);

- realloc()
  - Change size of block previous allocated by $malloc$() function
  - Possible of trying allocate or copy to new block of memory
  - void *realloc(void *ptr, size_t size);

Dynamic Memory Allocation Example

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

int main()
{
    char *str;

    /* Dynamic Memory allocation */
    str = (char *) malloc(50);
    if (str == NULL)
    {
        printf("Error in memory allocation.");
        return(1);
    }

    strcpy(str, "Programming is fun!");
    printf("String = %s\n", str);

    // Free the memory allocated
    free(str);

    return(0);
}
```

Dynamic Memory Allocation Example – alloc and free

```c
#include <stdio.h>
#include <stdlib.h> // For malloc and free

int main() {
    int n; // Number of elements for the array

    // Ask the user for the number of elements
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Dynamically allocate memory for n integers
    int *array = (int *)malloc(n * sizeof(int)); // malloc allocates memory
    if (array == NULL) { // Check if malloc failed
        printf("Memory allocation failed.\n");
        return 1; // Exit the program
    }

    // Fill the array with data
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &array[i]);
    }

    // Print the elements of the array
    printf("You entered:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(array);

    return 0;
}
```

```
Enter the number of elements: 5
Enter 5 integers:
1 2 3 4 5
You entered:
1 2 3 4 5
```

Dynamic Memory Allocation Example - calloc

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

int main() {
    int n = 3; // Number of elements

    // Allocate memory for 3 integers using calloc
    int *array = (int *)calloc(n, sizeof(int));

    if (array == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Show that calloc initializes memory to zero
    printf("Values in the array after calloc:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]); // Should print 0
    }
    printf("\n");

    // Free memory
    free(array);

    return 0;
}
```

```
Values in the array after calloc:
0 0 0
```

Dynamic Memory Allocation Example - realloc

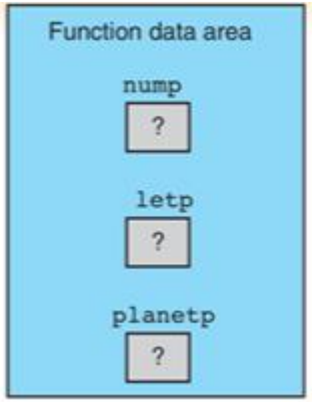
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n = 3;

    // Allocate memory for 3 integers using malloc
    int *array = (int *)malloc(n * sizeof(int));

    if (array == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Assign values to the array
    for (int i = 0; i < n; i++) {
        array[i] = i + 1; // Assign values 1, 2, 3
    }

    // Print initial values
    printf("Initial values in the array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
```

```c
/ Reallocate memory to hold 5 integers
    array = (int *)realloc(array, 5 * sizeof(int));
    if (array == NULL) {
        printf("Memory reallocation failed.\n");
        return 1;
    }

    // Assign values to the new elements
    for (int i = n; i < 5; i++) {
        array[i] = i + 1; // Assign values 4, 5
    }

    // Print updated values
    printf("Updated values in the array after realloc:\n");
    for (int i = 0; i < 5; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    // Free memory
    free(array);

    return 0;
}
```

```
Initial values in the array:
1 2 3
Updated values in the array after realloc:
1 2 3 4 5
```

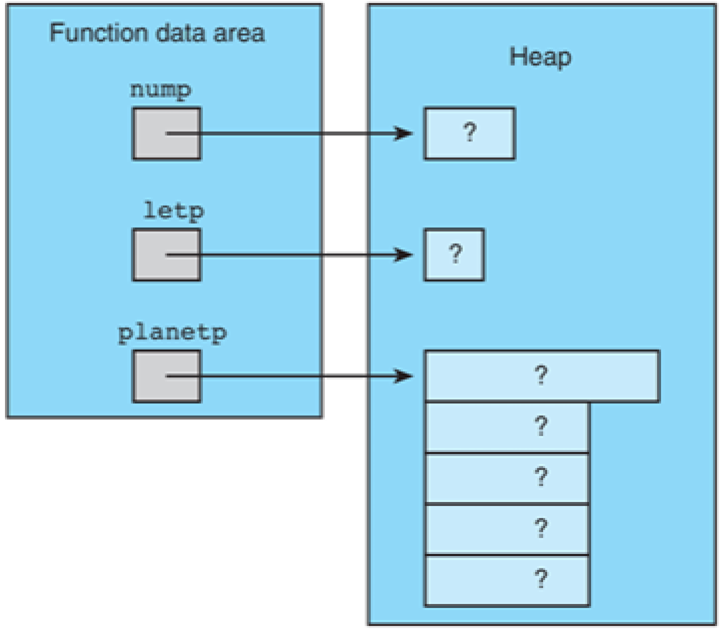
Memory Allocation (1)

```
int      *nump;
char     *letp;
planet_t *planetp;
```



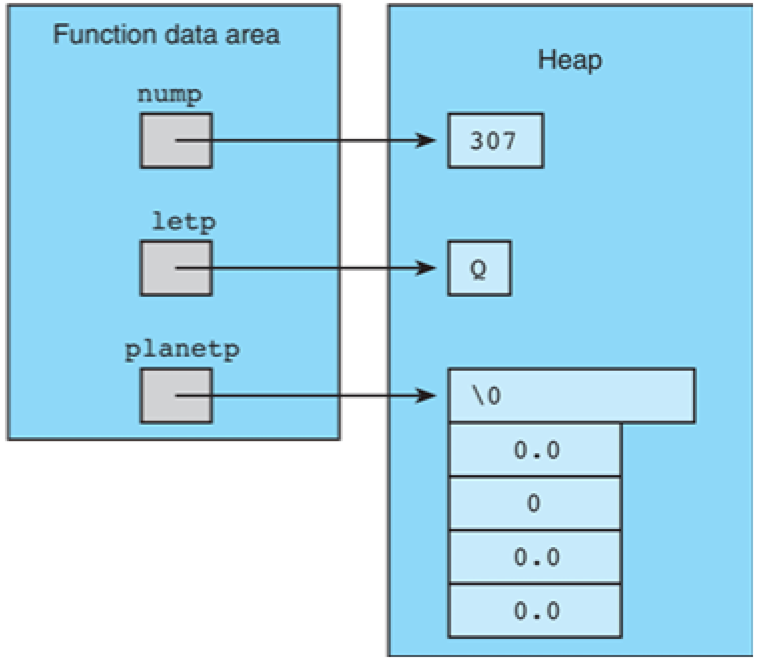
Memory Allocation (2)

```
nump = (int *)malloc(sizeof (int));
letp = (char *)malloc(sizeof (char));
planetp = (planet_t *)malloc(sizeof (planet_t));
```



Memory Allocation (3)

```
planet_t blank_planet = {"", 0, 0, 0, 0};
*nump = 307;
*letp = 'Q';
*planetp = blank_planet;
```



Dynamic Memory Allocation with calloc()

```c
#include <stdlib.h>  /* gives access to calloc */
int scan_planet(planet_t *plnp);

int
main(void)
{
      char      *string1;
      int       *array_of_nums;
      planet_t  *array_of_planets;
      int        str_siz, num_nums, num_planets, i;
      printf("Enter string length and string> ");
      scanf("%d", &str_siz);
      string1 = (char *)calloc(str_siz, sizeof (char));
      scanf("%s", string1);

      printf("\nHow many numbers?> ");
      scanf("%d", &num_nums);
```

Dynamic Memory Allocation with calloc()

```
        array_of_nums = (int *)calloc(num_nums, sizeof (int));
        array_of_nums[0] = 5;
        for (i = 1; i < num_nums; ++i)
              array_of_nums[i] = array_of_nums[i - 1] * i;

        printf("\nEnter number of planets and planet data> ");
        scanf("%d", &num_planets);
        array_of_planets = (planet_t *)calloc(num_planets,
                                              sizeof (planet_t));
        for (i = 0; i < num_planets; ++i)
              scan_planet(&array_of_planets[i]);
        . . .
}
```

```
Enter string length and string> 9 enormous

How many numbers?> 4

Enter number of planets and planet data> 2
Earth    12713.5 1  1.0 24.0
Jupiter 142800.0 4 11.9  9.925
```