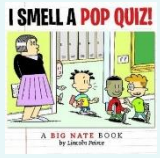
ITP20001/ECE 20010 Data Structures

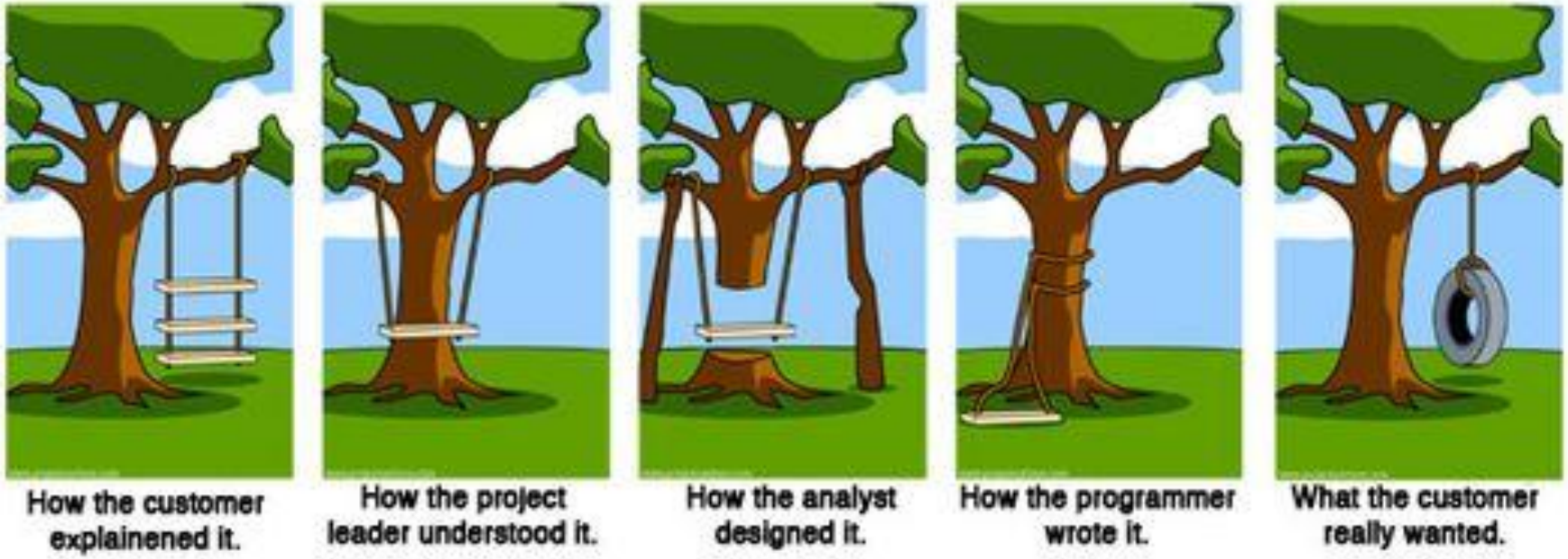
Data Structures

Chapter 2

- ***arrays***, *dynamically allocated arrays*
- ***structures*** *and unions in C*
- *strings*



Prof. Youngsup Kim, idebtor@handong.edu, Handong Global University



Original Source: http://www.tamingdata.com/2010/07/08/the-project-management-tree-swing-cartoon-past-and-present/

Is Python more powerful than C++?

Terry Lambert, Apple Core OS Kernel Team; technical lead on several projects over 8 years

Answered Mon

Python is capable of moving mountains.

Python is the most powerful computing language ever devised.

Python has an intuitive understanding of hardware that carries over into the person writing it.

Python is capable of banging registers without side effects.

Python is capable of representing register sets as structural objects with side effect for their access.

Python was written by some of the elder gods of computer programming.

Python is something I will be teaching my grandchildren, so that they can get a job, when everyone else has been replaced by robots.

Python is so amazing that...

Oh.

Wait.

20.9k Views · 288 Upvotes · Answer requested by Joe Smith

Upvote | 288 Downvote

Is Python more powerful than C++?

Terry Lambert, Apple Core OS Kernel Team; technical lead on several projects over 8 years

Answered Mon

Python is capable of moving mountains.

Python is the most powerful computing language ever devised.

Python has an intuitive understanding of hardware that carries over into the person writing it.

Python is capable of banging registers without side effects.

Python is capable of representing register sets as structural objects with side effect for their access.

Python was written by some of the elder gods of computer programming.

Python is something I will be teaching my grandchildren, so that they can get a job, when everyone else has been replaced by robots.

Python is so amazing that...

Oh.

Wait.

I meant C.

Not Python.

Sorry for any confusion.

Carry on.

#!$*! pointers...

20.9k Views · 288 Upvotes · Answer requested by Joe Smith

Upvote | 288 Downvote

Chapter 2 – Arrays and structures

2.1 Arrays

Array: Collections of data of the same type

Why arrays?

- **Efficient random access** (constant time) but **inefficient** insertion and deletion of elements.
- **Good locality** of reference when iterating through - much faster than iterating through (say) a linked list of the same size, which tends to jump around in memory.
- **Consequently**, arrays are most appropriate for storing a fixed amount of data which will be accessed in an unpredictable fashion.

Chapter 2 – Arrays and structures

2.1 Arrays

Array: Collections of data of the same type

ADT: Array is
objects: A set of pairs **<index, value>** where for each value of index there is value from the set item.
functions:
Array Create (j, list)
Item Retrieve(A, i)
Array Store(A, i, x)

Chapter 2 – Arrays and structures

2.1 Arrays

Array: Collections of data of the same type

Array example in C:

- **base address:** It is the address of the **first** element of an array which is **&list[0]** or **list**.
- **pointer arithmetic:** (ptr + 1) references to the **next element** of array regardless of its type.
- **dereferencing operator ***
  *(ptr + i) indicates **contents** of the (ptr + i) position of array.

Chapter 2 – Arrays and structures

2.1 Arrays

Code example: Program 2.1 (modified)

```
void main(void) {
  double array[] = {0, 1, 2, 3, 4};
  int n =

  printf("The sum is: %f\n",  sum(array, n));
  printf("The sum is: %f\n",  sumPointer(&array[0], n));
}
```

equivalent

```
double sum(double a[], int n)
{
  double total = 0;

  for (int i = 0; i < n; i++)
    total += a[i];
  return total;
}
```

```
double sumPointer(double a[], int n)
{
  double total = 0;

  for (int i = 0; i < n; i++)
    total +=
  return total;
}
```

Chapter 2 – Arrays and structures

2.1 Arrays

Code example: Program 2.1 (modified)

```
void main(void) {
  double array[] = {0, 1, 2, 3, 4};
  int n = sizeof(array) / sizeof(array[0]);

  printf("The sum is: %f\n",  sum(array, n));
  printf("The sum is: %f\n",  sumPointer(&array[0], n));
}
```

equivalent

```
double sum(double a[], int n)
{
  double total = 0;

  for (int i = 0; i < n; i++)
    total += a[i];
  return total;
}
```

```
double sumPointer(double a[], int n)
{
  double total = 0;

  for (int i = 0; i < n; i++)
    total +=
  return total;
}
```

Chapter 2 – Arrays and structures

2.1 Arrays

Code example: Program 2.1 (modified)

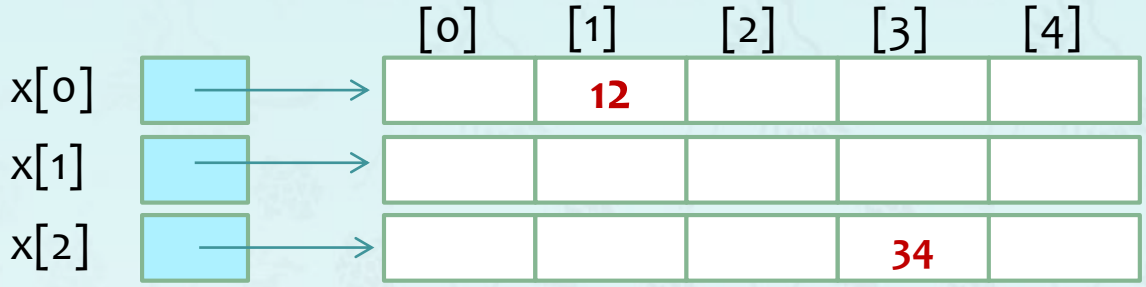
```
void main(void) {
  double array[] = {0, 1, 2, 3, 4};
  int n = sizeof(array) / sizeof(array[0]);

  printf("The sum is: %f\n",  sum(array, n));
  printf("The sum is: %f\n",  sumPointer(&array[0], n));
}
```

```
double sum(double a[], int n)
{
  double total = 0;

  for (int i = 0; i < n; i++)
    total += a[i];
  return total;
}
```

```
double sumPointer(double a[], int n)
{
  double total = 0;

  for (int i = 0; i < n; i++)
    total += *a++;
  return total;
}
```

2.1 Arrays

Two-dimensional array by example: int x[3][5]



- C finds the ***element x[i][j]*** by first accessing the point in x[i] and increment ***x[i]*** by ***j * sizeof(int)***.
  In your code, however, you just add ***x[i] + j***.

Chapter 2 – Arrays and structures

2.1 Arrays

Allocating two dimensional array:

Program 2.3 (enhanced) → Think about how to free!

```
int **make2dArray(int rows, int cols) {

  // Get memory for row pointers
  int **a =          malloc(rows * sizeof(       );




}
```

Chapter 2 – Arrays and structures

2.1 Arrays

Allocating two dimensional array:

Program 2.3 (enhanced) → Think about how to free!

```
int **make2dArray(int rows, int cols) {

  // Get memory for row pointers
  int **a = (int **) malloc(rows * sizeof(int *));




}
```

2.1 Arrays

Allocating two dimensional array:

Program 2.3 (enhanced) → Think about how to free!

```
int **make2dArray(int rows, int cols) {

  // Get memory for row pointers
  int **a = (int **) malloc(rows * sizeof(int *));
  assert(a != NULL);



}
```

a must: error checking after malloc()

2.1 Arrays

Allocating two dimensional array:

Program 2.3 (enhanced) → Think about how to free!

```
int **make2dArray(int rows, int cols) {

  // Get memory for row pointers
  int **a = (int **) malloc(rows * sizeof(int *));
  assert(a != NULL);

  for (int i = 0; i < rows; i++)  // for each row
    a[i] = (          ) malloc(                    );
  return a;
}
```

a must: error checking after malloc()

assert(a[i] != NULL);

2.1 Arrays

Allocating two dimensional array:

Program 2.3 (enhanced) → Think about how to free!

```
int **make2dArray(int rows, int cols) {

  // Get memory for row pointers
  int **a = (int **) malloc(rows * sizeof(int *));
  assert(a != NULL);

  for (int i = 0; i < rows; i++)  // for each row
    a[i] = (int *) malloc(cols * sizeof(int));
  return a;
}
```

a must: error checking after malloc()

assert(a[i] != NULL);

Q: Can you tell how many times malloc() is invoked? Remember this fact.

Chapter 2 – Arrays and structures

2.1 Arrays

Allocating two dimensional array:

Program: invoking and using 2d array

```
int  **xyArray;
xyArray = make2dArray (3, 5);
xyArray[0][1] = 12;
xyArray[2][3] = 34;
```

Program 2.3 (enhanced) → Think about how to free!

```
int **make2dArray(int rows, int cols) {

  // Get memory for row pointers
  int **a = (int **) malloc(rows * sizeof(int *));
  assert(a != NULL);

  for (int i = 0; i < rows; i++)  // for each row
    a[i] = (int *) malloc(cols * sizeof(int));
  return a;
}
```

a must: error checking after malloc()

assert(a[i] != NULL);

Chapter 2 – Arrays and structures

2.1 Arrays

Program (driver for make2dArray)

```
int main(int argc, char *argv[]) {
    int rows = 3, cols = 5;
    int i, j, **xyArray;

    xyArray = make2dArray(rows, cols);

    xyArray[0][1] = 12;
    xyArray[2][3] = 34;

    print2dArray(xyArray, rows, cols);
    free2dArray(xyArray, rows);
}
```

Q: Any potential problems in the code above?

Chapter 2 – Arrays and structures

2.1 Arrays

Program (driver for make2dArray)

```
int main(int argc, char *argv[]) {
    int rows = 3, cols = 5;
    int i, j, **xyArray;

    xyArray = make2dArray(rows, cols);

    xyArray[0][1] = 12;
    xyArray[2][3] = 34;

    print2dArray(xyArray, rows, cols);
    free2dArray(xyArray, _______________);
}
```

Q: Any potential problems in the code above?

1: xyArray has not been initialized in neither make2dArray() nor main().

2: No error checking after make2dArray() function call.

Chapter 2 – Arrays and structures

2.1 Arrays

Program (2d array free example)

```
void free2dArray(int **a, ____________________) {



}
```

- Free the resources after use! Otherwise, "**Memory leaks**!"
- Invoke free() as much as malloc() was invoked.

Program 2.3 (enhanced) → Think about how to free!

```
int **make2dArray(int rows, int cols) {

  // Get memory for row pointers
  int **a = (int **) malloc(rows * sizeof(int *));
  assert(a != NULL);

  for (int i = 0; i < rows; i++)  // for each row
    a[i] = (int *) malloc(cols * sizeof(int));
  return a;
}
```

Chapter 2 – Arrays and structures

2.1 Arrays

Program (2d array free example)

```
void free2dArray(int **a, ____________) {



}
```

- ❖ Free the resources after use! Otherwise, "**Memory leaks**!"
- ❖ Invoke free() as much as malloc() was invoked.

Q: Can you tell how many times malloc() was invoked?

Chapter 2 – Arrays and structures

2.1 Arrays

Program (2d array free example)

```
void free2dArray(int **a, int rows) {
  // free each row first




}
```

- Free the resources after use! Otherwise, "**Memory leaks**!"
- Invoke free() as much as malloc() was invoked.

Q: Can you tell how many times malloc() was invoked?

Chapter 2 – Arrays and structures

2.1 Arrays

Program (2d array free example)

```
void free2dArray(int **a, int rows) {
  // free each row first
  for (int i = 0; i < rows; i++)



}
```

- Free the resources after use! Otherwise, "**Memory leaks**!"
- Invoke free() as much as malloc() was invoked.

Q: Can you tell how many times malloc() was invoked?

2.1 Arrays

Program (2d array free example)

```
void free2dArray(int **a, int rows) {
  // free each row first
  for (int i = 0; i < rows; i++)
    free(a[i]);   // free(*(a+i));


}
```

- Free the resources after use! Otherwise, "**Memory leaks**!"
- Invoke free() as much as malloc() was invoked.

Q: Can you tell how many times malloc() was invoked?

Chapter 2 – Arrays and structures

2.1 Arrays

Program (2d array free example)

```
void free2dArray(int **a, int rows) {
  // free each row first
  for (int i = 0; i < rows; i++)
    free(a[i]);   // free(*(a+i));

  free(a);
}
```

- Free the resources after use! Otherwise, "**Memory leaks**!"
- Invoke free() as much as malloc() was invoked.

Q: Can you tell how many times malloc() was invoked?

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **C has no objects,** but has data structures that can help fill in roles. These are called "**struct**" and "**union**", and contain **data** only .
- These are not true objects b/c they can't contain _____ .

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **struct as a type, and initialization**

struct – user defined

```
struct Student {
    char name[32];
    int     SN;
};
```

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **struct as a type, and initialization**

struct – user defined

```
struct Student {
    char name[32];
    int     SN;
};

struct Student ur = {"Lee", 20};
struct Student my;
strcpy(my.name, "Kim"};
```

not a variable, but a type

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **struct as a type, and initialization**

struct – user defined

```
struct Student {
    char name[32];
    int     SN;
};

struct Student ur = {"Lee", 20};
struct Student my;
strcpy(my.name, "Kim"};
```

not a variable, but a type

❖ **Conclusion:**

You can create your own structure data types by using the **typedef**.

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Example: user defined data type using "typedef"**

struct – user defined

```
struct Student {
   char name[32];
   int    SN;
};
struct Student my = {"Kim",20};

strcpy(my.name, "James");
my.SN = 20;
```

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
}student;
Student ur = {"Lee", 30}, my = {"Kim", 20};

if (strcmp(ur.name, my.name))
    printf ("different name\n");
else
    printf("same name\n");
ur = my;
```

- ANSI C permits **structure assignment**, most earlier versions of C do not.!
- Structures must begin and end on the same type of memory boundary – 4, 8 or 16.

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

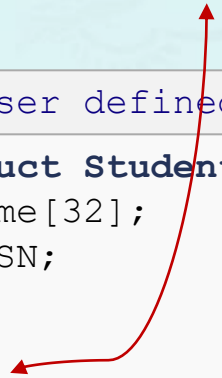
- **Struct pointers:**

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**



typedef – user defined

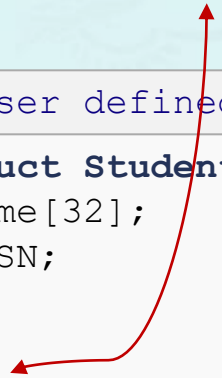
```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (                )malloc(                );
```

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**



typedef – user defined

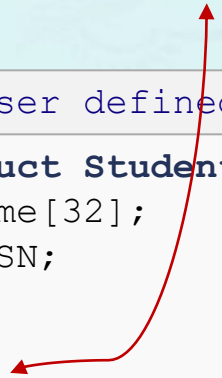
```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
```

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**



typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
```

Q: Then, how do you access the structure members with struct pointer?

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
strcpy ((*my).name, "Kim");
(*my).SN = 20;    // (*my).SN
```

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
strcpy (my->name, "Kim");
my->SN = 20;      // (*my).SN
```

- With pointers to a structure, use → the **dereference operator** to access members**, exclusively.**

Q: How can you do a structure copy (or copying ***ur*** into ***my***)?
Q: Why is it called "**dereference** operators"?

A: *my =ur;
A: my->name
is, for all purposes, equivalent to :
(*my).name ;

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
strcpy (my->name, "Kim");
my->SN = 20;      // (*my).SN
```

❖ Let's go one more step!
How about redefining
student *my
since we are going to love the pointer!

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
strcpy (my->name, "Kim");
my->SN = 20;      // (*my).SN
```

❖ Let's go one more step!
How about redefining
student *my
since we are going to love the pointer!

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;
typedef struct Student *pStudent;
```

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
strcpy (my->name, "Kim");
my->SN = 20;      // (*my).SN
```

❖ Let's go one more step!
How about redefining
student *my
since we are going to love the pointer!

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;
typedef struct Student *pStudent;

pStduent my;           // Student *my;
Student ur = {"Lee", 25};

my = (pStduent)malloc(sizeof(Student));
strcpy (my->name, "Kim");
my->SN = 20;           // (*my).SN
```

Chapter 2 – Arrays and structures

2.3 Structures and unions

Struct – a way to organize data of the different types.

- **Struct pointers:**

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;

Student  *my,   ur = {"Lee", 25};

my = (Student *)malloc(sizeof(Student));
strcpy (my->name, "Kim");
my->SN = 20;      // (*my).SN
```

❖ Let's go one more step!
How about redefining
student *my
since we are going to love the pointer!

typedef – user defined

```
typedef struct Student{
    char name[32];
    int    SN;
} Student;
typedef struct Student *pStudent;

pStduent my;           // Student *my;
Student ur = {"Lee", 25};

my = (pStduent)malloc(sizeof(Student));
strcpy (my->name, "Kim");
my->SN = 20;           // (*my).SN
```

Example: Clock(1)

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK *clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

What is wrong in the code?

Example: Clock(1)

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

Example: Clock(2)

Rewrite the code using a pointer `*ptr and malloc()` instead of `CLOCK clock;`

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

Example: Clock(2)

Rewrite the code using a pointer `*ptr and malloc()` instead of `CLOCK clock;`

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);

int main (void) {
   int i = 0;    // malloc to create a clock




   for(i = 0; i < 6; ++i) {
      increment (&ptr);
      show (&ptr);
   }
   return 0;
}
```

Example: Clock(2)

Rewrite the code using a pointer `*ptr and malloc()` instead of `CLOCK clock;`

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);

int main (void) {
   int i = 0;
   CLOCK *ptr =
          (CLOCK *)malloc(sizeof(CLOCK));



   for(i = 0; i < 6; ++i) {
      increment (&ptr);
      show (&ptr);
   }
   return 0;
}
```

Still has a problem?

Example: Clock(2)

Rewrite the code using a pointer `*ptr and malloc()` instead of `CLOCK clock;`

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);

int main (void) {
   int i = 0;
   CLOCK *ptr =
          (CLOCK *)malloc(sizeof(CLOCK));
   ptr->hr = 14;
   ptr->min = 38;
   ptr->sec = 56;
   for(i = 0; i < 6; ++i) {
      increment (&ptr);
      show (&ptr);
   }
   return 0;
}
```

Still has a problem?

Example: Clock(2)

Rewrite the code using a pointer `*ptr and malloc()` instead of `CLOCK clock;`

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);

int main (void) {
   int i = 0;
   CLOCK *ptr =
          (CLOCK *)malloc(sizeof(CLOCK));
   ptr->hr = 14;
   ptr->min = 38;
   ptr->sec = 56;
   for(i = 0; i < 6; ++i) {
      increment (ptr);
      show (ptr);
   }
   return 0;
}
```

Example: Clock(1,2)

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

```
// increment the time by one second.
void increment (CLOCK *ptr) {
   ptr->sec++;

   if (ptr->sec == 60){
      ptr->sec = 0;
      ptr->min++;

      if (ptr->min == 60){
         ptr->min = 0;
         ptr->hr++;
         if (ptr->hr == 24) ptr->hr = 0;
      }
   }
}

// show the current time in military form.
void show (CLOCK *ptr)  {
   printf("%02d:%02d:%02d\n",
        ptr->hr, ptr->min, ptr->sec);
}
```

Example: Clock(3)

Rewrite the code using **typedef of a pointer to a struct**

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};


void increment (CLOCK *ptr);
void show      (CLOCK *ptr);

int main (void) {
   int i = 0;
   CLOCK *ptr =
      (CLOCK *)malloc(sizeof(CLOCK));
   ptr->hr = 14;
   ptr->min = 38;
   ptr->sec = 56;
   for(i = 0; i < 6; ++i) {
      increment (ptr);
      show (ptr);
   }
   return 0;
}
```

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
}CLOCK;
typedef struct CLOCK* pClock;
```

Example: Clock(3)

Rewrite the code using **typedef of a pointer to a struct**

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};


void increment (CLOCK *ptr);
void show      (CLOCK *ptr);

int main (void) {
   int i = 0;
   CLOCK *ptr =
      (CLOCK *)malloc(sizeof(CLOCK));
   ptr->hr = 14;
   ptr->min = 38;
   ptr->sec = 56;
   for(i = 0; i < 6; ++i) {
      increment (ptr);
      show (ptr);
   }
   return 0;
}
```

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
}CLOCK;
typedef struct CLOCK* pClock;

void increment (pCLOCK ptr);
void show      (pCLOCK ptr);
```

Example: Clock(3)

Rewrite the code using **typedef of a pointer to a struct**

```c
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};


void increment (CLOCK *ptr);
void show      (CLOCK *ptr);

int main (void) {
   int i = 0;
   CLOCK *ptr =
      (CLOCK *)malloc(sizeof(CLOCK));
   ptr->hr = 14;
   ptr->min = 38;
   ptr->sec = 56;
   for(i = 0; i < 6; ++i) {
      increment (ptr);
      show (ptr);
   }
   return 0;
}
```

```c
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
}CLOCK;
typedef struct CLOCK* pClock;

void increment (pCLOCK ptr);
void show      (pCLOCK ptr);

int main (void) {
   int i = 0;
   pCLOCK ptr =
      (pCLOCK)malloc(sizeof(CLOCK));
   ptr->hr = 14;
   ptr->min = 38;
   ptr->sec = 56;
   for(i = 0; i < 6; ++i) {
      increment (ptr);
      show (ptr);
   }
   return 0;
}
```

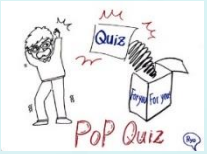
Example: Clock(3)

Rewrite the code using **typedef of a pointer to a struct**

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
};

void increment (CLOCK *ptr);
void show      (CLOCK *ptr);


int main (void) {
   int i = 0;

   CLOCK clock = {14, 38, 56};

   for(i = 0; i < 6; ++i) {
      increment (&clock);
      show (&clock);
   }
   return 0;
}
```

```
#include <stdio.h>

typedef struct CLOCK{
   int hr, min, sec;
}CLOCK;
typedef struct CLOCK* pClock;

void increment (pCLOCK ptr);
void show      (pCLOCK ptr);

int main (void) {
   int i = 0;
   pCLOCK ptr =
      (pCLOCK)malloc(sizeof(CLOCK));
   ptr->hr = 14;
   ptr->min = 38;
   ptr->sec = 56;
   for(i = 0; i < 6; ++i) {
      increment (ptr);
      show (ptr);
   }
   return 0;
}
```

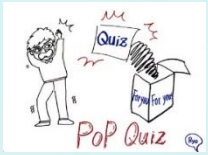
Chapter 2 – Arrays and structures



2.3 Self-referenced structures

Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
```

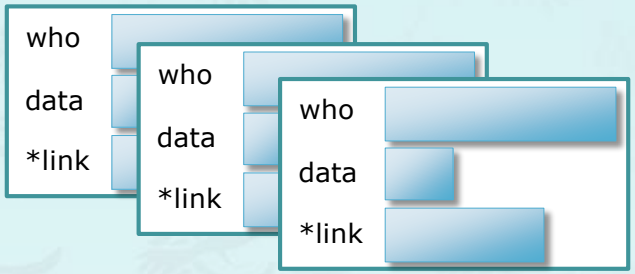


2.3 Self-referenced structures

Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
```

list a, b, c;

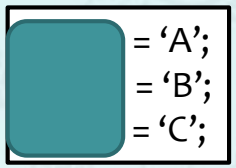


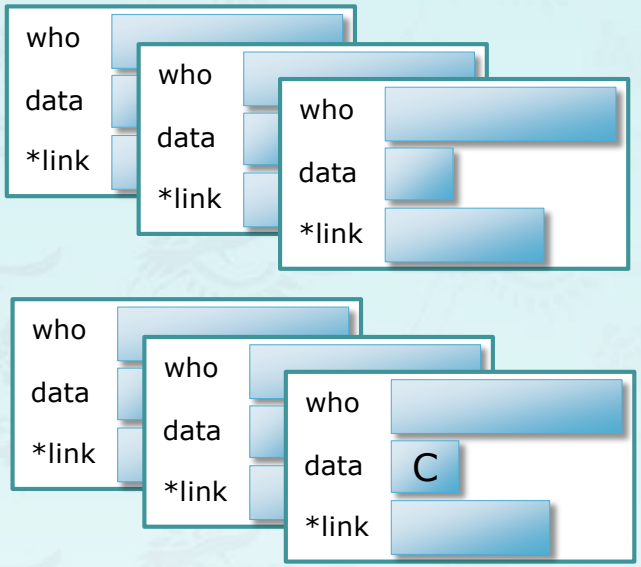
2.3 Self-referenced structures

Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
```

list a, b, c;



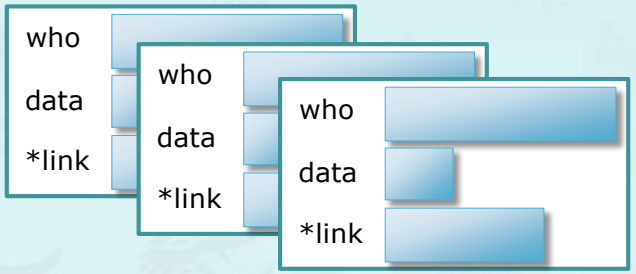


2.3 Self-referenced structures

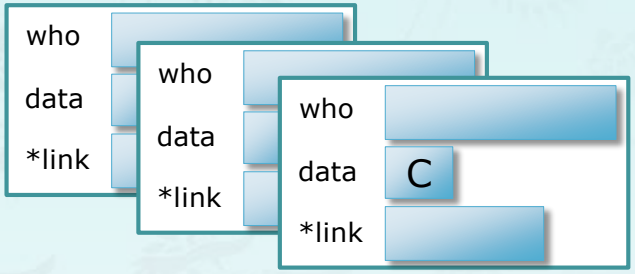
Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
a.data = 'A';
b.data = 'B';
c.data = 'C';
```

list a, b, c;




```
a.data = 'A';
b.data = 'B';
c.data = 'C';
```

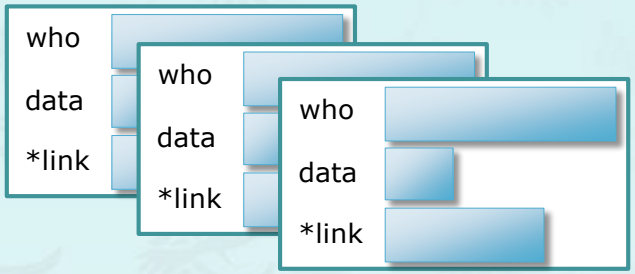


2.3 Self-referenced structures

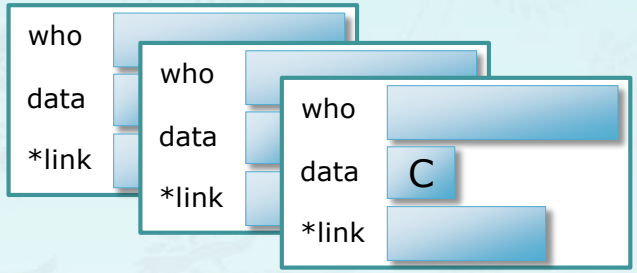
Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
a.data = 'A';
b.data = 'B';
c.data = 'C';
```

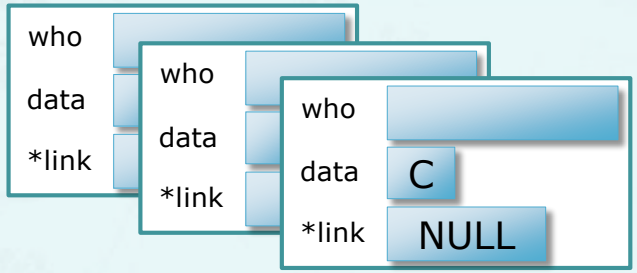
list a, b, c;




```
a.data = 'A';
b.data = 'B';
c.data = 'C';
```



= NULL;

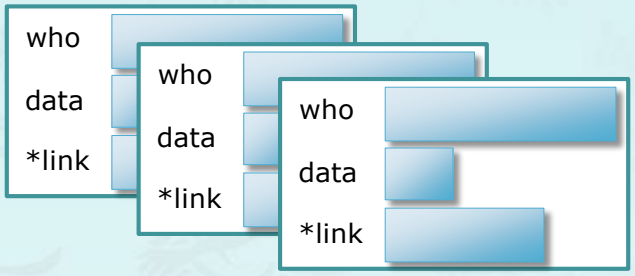


2.3 Self-referenced structures

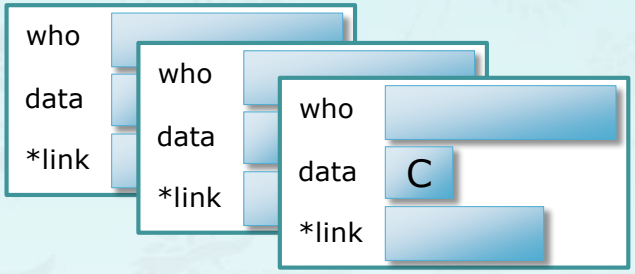
Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
a.data = 'A';
b.data = 'B';
c.data = 'C';
a.link = b.link = c.link = NULL;
```

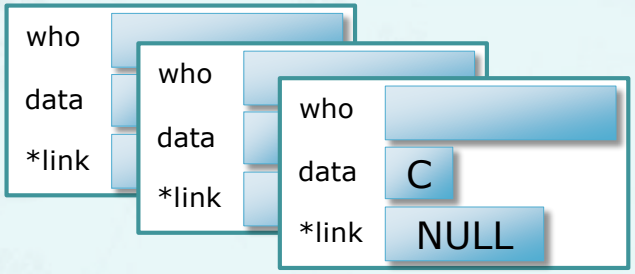
list a, b, c;



a.data = 'A';
b.data = 'B';
c.data = 'C';



a.link = b.link = c.link = NULL;

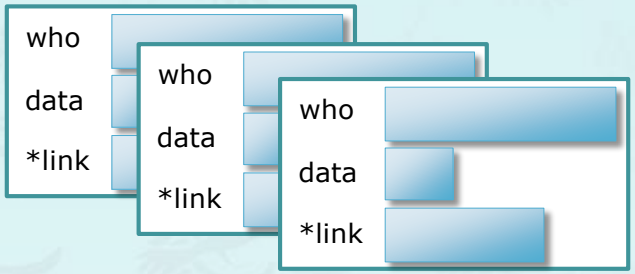


2.3 Self-referenced structures

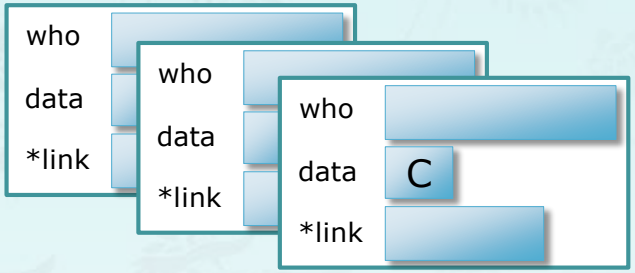
Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
a.data = 'A';
b.data = 'B';
c.data = 'C';
a.link = b.link = c.link = NULL;
```

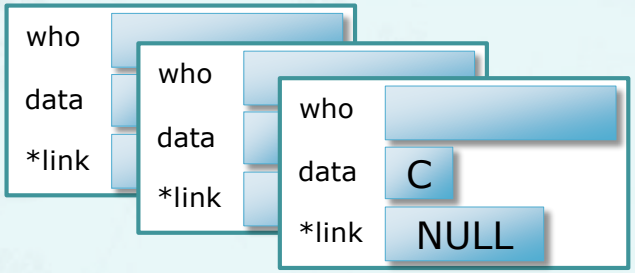
list a, b, c;

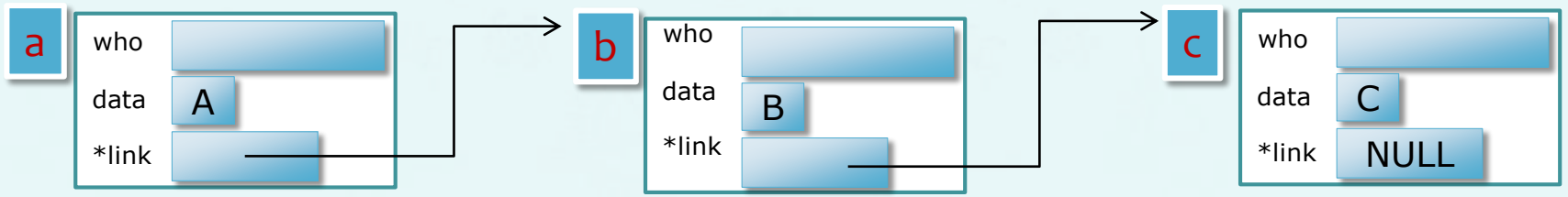


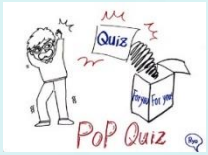
a.data = 'A';
b.data = 'B';
c.data = 'C';



a.link = b.link = c.link = NULL;





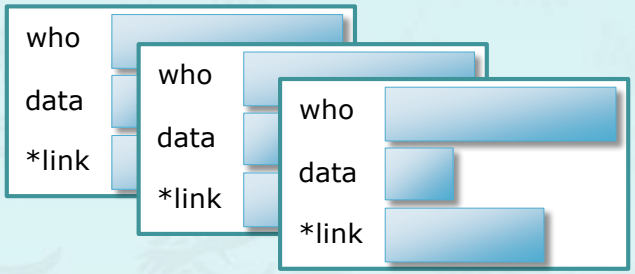


2.3 Self-referenced structures

Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
list *p, *q, *r;
```

list a, b, c;



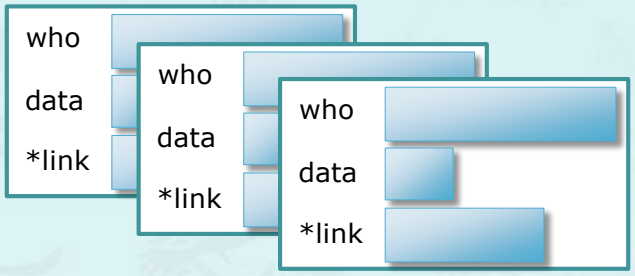
Chapter 2 – Arrays and structures

2.3 Self-referenced structures

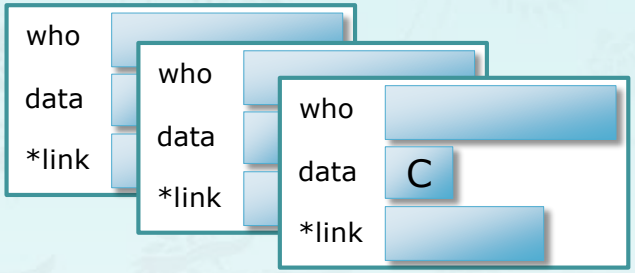
Exercise: Link a, b and c nodes;

```
typedef struct list {
      student who;
      char  data;
      list    *link;      // pointer to list type
}list;

list a, b, c;
list *p, *q, *r;
```

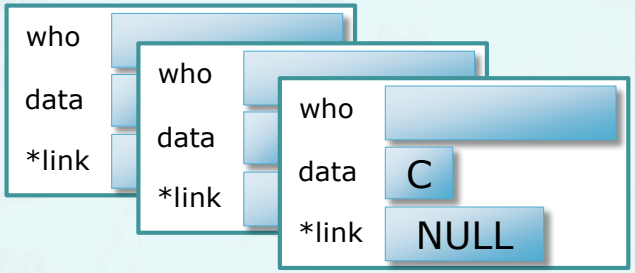
list a, b, c;



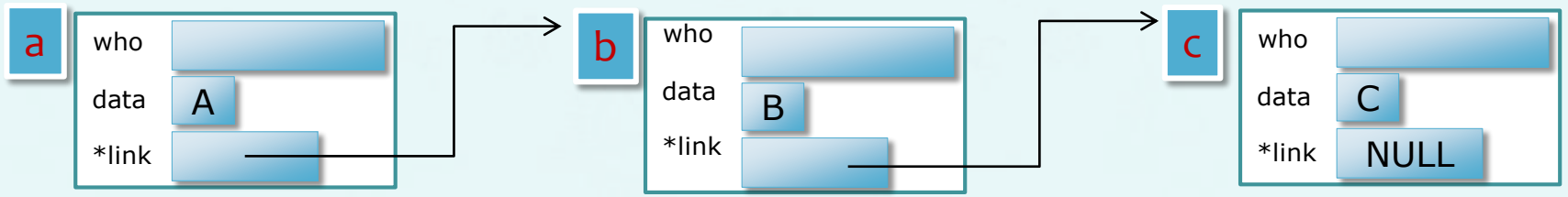

```
a.data = 'A';
b.data = 'B';
c.data = 'C';
```



a.link = b.link = c.link = NULL;



(1) Let each p, q, and r points to a, b, and c;
(2) Store each 'a', 'b', and 'c' in data, and NULL in link, respectively.
(3) Connect them using p, q and r as shown below:

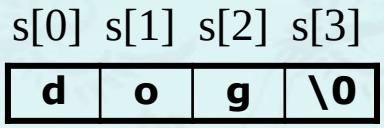


2.7 Strings

String: a character array terminated with the null \0 in C.

Example:

char **s**[]={"dog"};

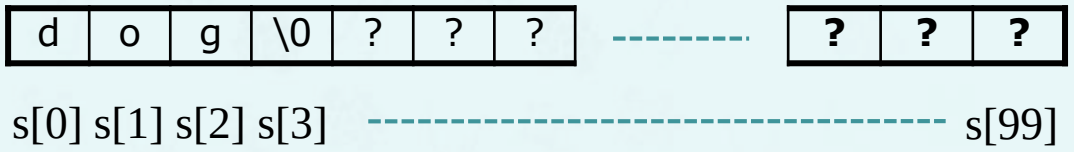


char t[]={"house"};

- The index of the char array begins with 0 always.
- The null \0 is included in the character array by the compiler.
- The string function strlen(**s**) above returns 3 .
- The string functions are defined in <string.h>.

char s[100]={"dog"}; // compiler allocates 4 or 100 chars or 101?
// strlen(s) = ?

s[] in memory:



2.7 Strings

- **Figure 2.8** - a brief summary of C string functions.
  - fix **some typos**
  - use #include <string.h>
- **Caution**:
  Some string functions change the input parameter (*char *dest*)
  strcat, strncat, strcpy, and **strncpy.**
  They do not check for sufficient space in *(char *dest)*, it is therefore a potential cause of **buffer overruns** which is very critical.
- MS VS C/C++ compiler checks these (CRT – C Run Time) functions and recommend the security enhance version (_s) such as **strcat_s, strcpy_s** etc.
- To suppress this errors and warnings, **however**, you must set in Project → Properties → C/C++ → General → CRT Checks – **No.**
  or you may add **_CRT_SECURE_NO_WARNINGS** in C/C++ → Preprocessor → Preprocessor Definition

2.7 Strings

❖ **Figure 2.8** - a brief summary of C string functions.
- fix **some typos**
- use #include <string.h>

Exercise: What is the output?

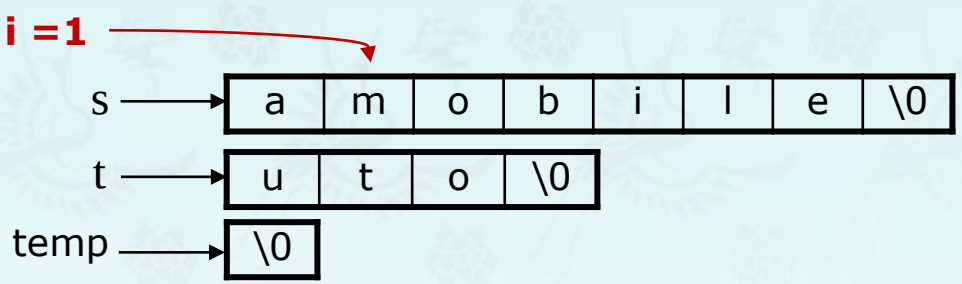
```
char myStr[100] = {"God is good "};
char urStr[] = {"all the time!"};
printf("%s\n", strcat(myStr, urStr));
printf("%s %s\n", myStr, urStr);
```

2.7 Strings

Example: String insertion(Figure 2.10)
Insert **t** into **s**, starting at the **i** position of **s**.

```
void strnins(char *s, const char *t, int i)
```

initially

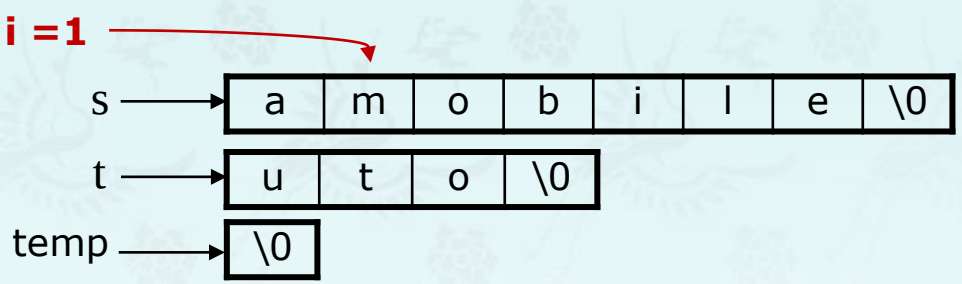


2.7 Strings

Example: String insertion(Figure 2.10)
Insert **t** into **s**, starting at the **i** position of **s**.

```
void strnins(char *s, const char *t, int i)
```

initially



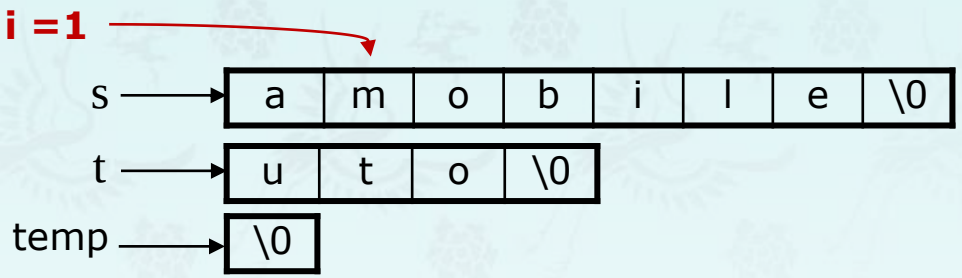
after strncpy(temp, s, i)

2.7 Strings

Example: String insertion(Figure 2.10)
Insert **t** into **s**, starting at the **i** position of **s**.

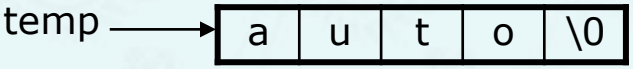
```
void strnins(char *s, const char *t, int i)
```

initially



after strncpy(temp, s, i)

after strcat(temp, t)

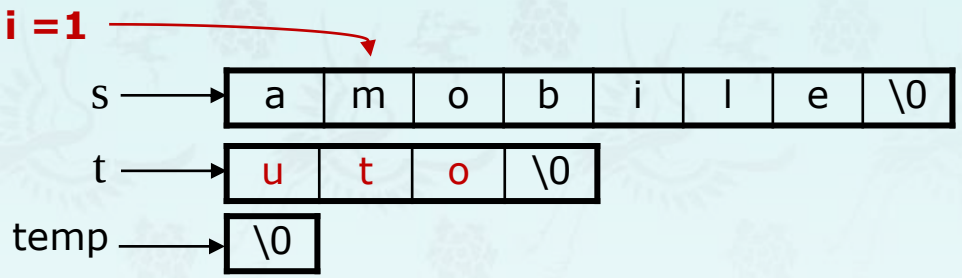


2.7 Strings

Example: String insertion(Figure 2.10)
Insert **t** into **s**, starting at the **i** position of **s**.

```
void strnins(char *s, const char *t, int i)
```

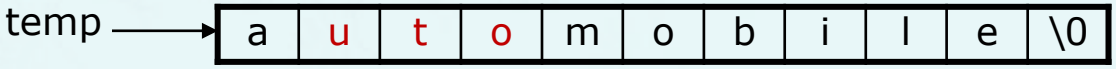
initially



after strncpy(temp, s, i)

after strcat(temp, t)

after strcat (temp,)



2.7 Strings

Example: String insertion -
Insert the string **t** into string **s**, starting at the **i**th position of string **s**.

```
void strnins(char *s, char *t, int i)
```

Program **2.12 strnins() buggy**→HSet 04

```
void strnins(char *s, char *t, int i) {
  char string[MAX_SIZE], *temp = string;
  if ((i < 0) || (i > strlen(s)))
    printf("Position is out of bounds \n")
  else {
    if (!(strlen(s)))
      strcpy(s, t);
    else if (strlen(t)) {
      strncpy(temp, s, i);
      strcat(temp, t);
      strcat(temp, (s + i));
      strcpy(s, temp);
    }
  }
}
```

Additionally there is room to improve.

- What is a shortcoming of this function?
- Does it following the coding standard such as readability, naming, return type?
- Is there any extra variable or cost of computing?

ECE 20010 Data Structures

Data Structures

Chapter 2

- ***arrays***, *dynamically allocated arrays*
- ***structures*** *and unions in C*
- *stringsMath review*

Prof. Youngsup Kim, idebtor@handong.edu, Handong Global University


Summary

&


quaestio quaestio qo ? ? ? ?