Chapter 6 structures

Speaker: Lung-Sheng Chien

OutLine

- Basics of structures
  - usage
  - heterogeneous aggregation
  - padding and alignment
- Structures and functions
- Arrays of structures
- Self-referential structure

Structure representation of 2D point [1]

point is called structure tag

```
#include <stdio.h>

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30 } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3

    printf("pt = (%d, %d)\n", pt.x , pt.y );
    printf("maxpt = (%d, %d)\n", maxpt.x , maxpt.y );

    printf("size of structure point = %d\n", sizeof( struct point) ) ;

    return 0 ;
}
```

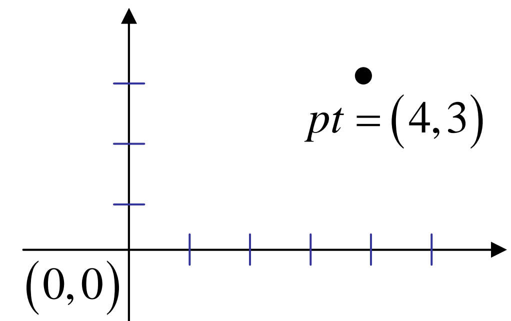
x and ***y*** in structure are called members

declare variable ***pt*** of type structure point, but members ***x*** and ***y*** are not set.

Initilize ***maxpt*** as ***x*** = 20 and ***y*** = 30 according to order of x and y in type *structure point*

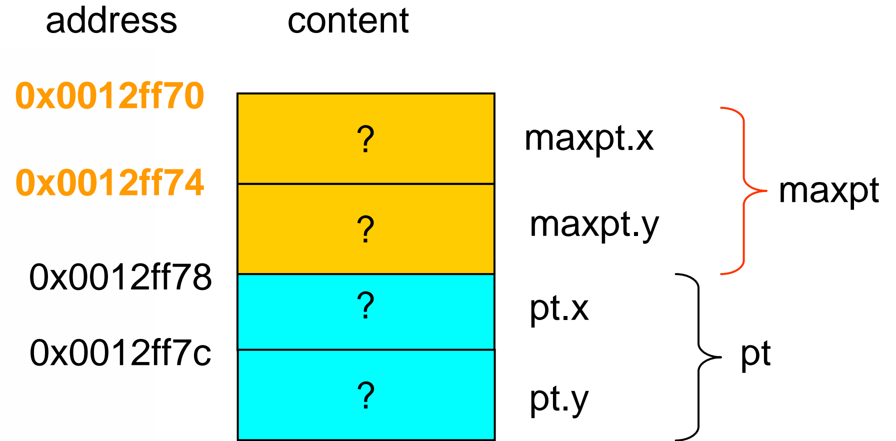
Use dot operator **.** to access member ***x*** of structure point

Size of *structure point* = sizeof(***x***) + sizeof(***y***)



Structure representation of 2D point [2]

```
#include <stdio.h>

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30 } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3
```



Context: main(int, char

Name	Value
argc	1
⊞ argv	0x003720a0
⊞ maxpt	{...}
⊞ pt	{...}

in(in
inCRT
RNEL3

Name	Value
⊟ &pt	0x0012ff78
x	-858993460
y	-858993460
⊞ &pt.x	0x0012ff78
⊞ &pt.y	0x0012ff7c
⊟ &maxpt	0x0012ff70
x	-858993460
y	-858993460
⊞ &maxpt.x	0x0012ff70
⊞ &maxpt.y	0x0012ff74
⊟ pt	{...}
x	-858993460
y	-858993460

按 F10

Unknown value since we don’t set them

Structure representation of 2D point [3]

```
#include <stdio.h>

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30 } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3
```

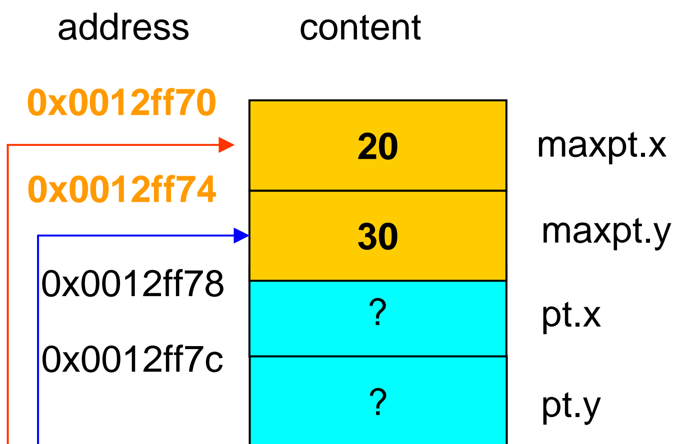
Context: main(int, char

Name	Value
⊞ maxpt	{...}
⊞ pt	{...}
pt.x	-858993460

in(in
inCRT
RNEL3

Name	Value
⊟ &pt	0x0012ff78
x	-858993460
y	-858993460
⊞ &pt.x	0x0012ff78
⊞ &pt.y	0x0012ff7c
⊟ &maxpt	0x0012ff70
x	20
y	30
⊞ &maxpt.x	0x0012ff70
⊞ &maxpt.y	0x0012ff74
⊟ pt	{...}
x	-858993460
y	-858993460

按 F10



Structure representation of 2D point [4]

```c
struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30 } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3

    printf("pt = (%d, %d)\n", pt.x , pt.y );
    printf("maxpt = (%d, %d)\n", maxpt.x , maxpt.y );
```

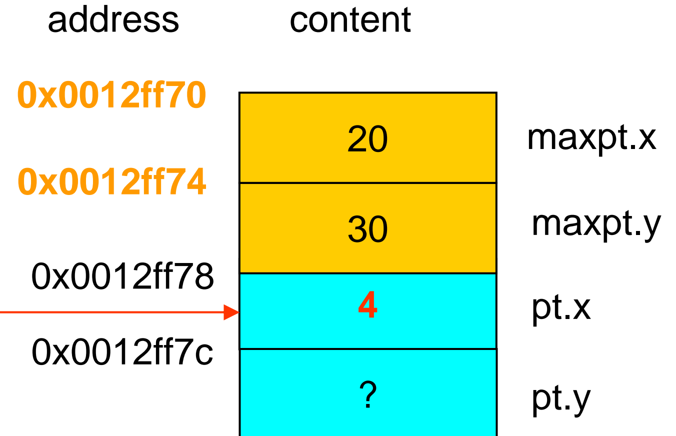
Context: main(int, char

Name	Value
pt.x	4
pt.y	-858993460

in(in
inCRT
RNEL3

Name	Value
&pt	0x0012ff78
x	4
y	-858993460
&pt.x	0x0012ff78
&pt.y	0x0012ff7c
&maxpt	0x0012ff70
x	20
y	30
&maxpt.x	0x0012ff70
&maxpt.y	0x0012ff74
pt	{...}
x	4
y	-858993460

按 F10



Structure representation of 2D point [5]

```
struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30 } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3

    printf("pt = (%d, %d)\n", pt.x , pt.y );
    printf("maxpt = (%d, %d)\n", maxpt.x , maxpt.y );
```

Context: main(int, char

Name	Value
pt.x	4
pt.y	3

in(in
inCRT
RNEL3

Name	Value
&pt	0x0012ff78
x	4
y	3
&pt.x	0x0012ff78
&pt.y	0x0012ff7c
&maxpt	0x0012ff70
x	20
y	30
&maxpt.x	0x0012ff70
&maxpt.y	0x0012ff74
pt	{...}
x	4
y	3

address	content	
0x0012ff70	20	maxpt.x
0x0012ff74	30	maxpt.y
0x0012ff78	4	pt.x
0x0012ff7c	3	pt.y

structure v.s. array [1]

```c
#include <stdio.h>

int main( int argc, char* argv[] )
{
    int  pt[2] ;
    int  maxpt[2] = { 20, 30 } ;

    pt[0] = 4 ; // set x component of point pt as 4
    pt[1] = 3 ; // set y component of point pt as 3

    printf("pt = (%d, %d)\n", pt[0] , pt[1] );
    printf("maxpt = (%d, %d)\n", maxpt[0] , maxpt[1] );

    return 0 ;
}
```

address	content	
0x0012ff70	20	maxpt[0]
0x0012ff74	30	maxpt[1]
0x0012ff78	?	pt[0]
0x0012ff7c	?	pt[1]

pt.x → pt[0]

pt.y → pt[1]

maxpt.x → maxpt[0]

maxpt.y → maxpt[1]

```c
int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30 } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3

    printf("pt = (%d, %d)\n", pt.x , pt.y );
    printf("maxpt = (%d, %d)\n", maxpt.x , maxpt.y );

    printf("size of structure point = %d\n", sizeof( struct point) ) ;

    return 0 ;
}
```

Question: why not use array?

structure v.s. array [2]
add a new field into structure

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

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
    char name[6] ; // name of the point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30, "Earth" } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3
    strcpy( pt.name, "Venus") ; // set name of pt

    printf("pt = (%d, %d, %s )\n", pt.x , pt.y, pt.name );
    printf("maxpt = (%d, %d, %s )\n", maxpt.x , maxpt.y, maxpt.name );

    printf("size of structure point = %d\n", sizeof( struct point) ) ;

    return 0 ;
}
```

; is necessary

1. add a field, character string, named ***name***

2. add one more initilization field, string copy is done by compiler

3. Use ***strcpy*** to set ***name*** field of ***pt***

4. add one more output filed

Advantage of structure: aggregation of heterogeneous data type

Question: How can you do when you use array to implement?

Padding and Alignment of structure [1]

```c
struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
    char name[6] ; // name of the point
} ;

int main( int argc, char* argv[] )
{
    struct point  pt ;
    struct point  maxpt = { 20, 30, "Earth" } ;

    pt.x = 4 ; // set x component of point pt as 4
    pt.y = 3 ; // set y component of point pt as 3
    strcpy( pt.name, "Venus") ; // set name of pt

    printf("pt = (%d, %d, %s )\n", pt.x , pt.y, pt.name );
```

Context: main(int, cha

Name	Value
argc	1
⊞ argv	0x003720a(
⊞ maxpt	{...}
⊞ pt	{...}

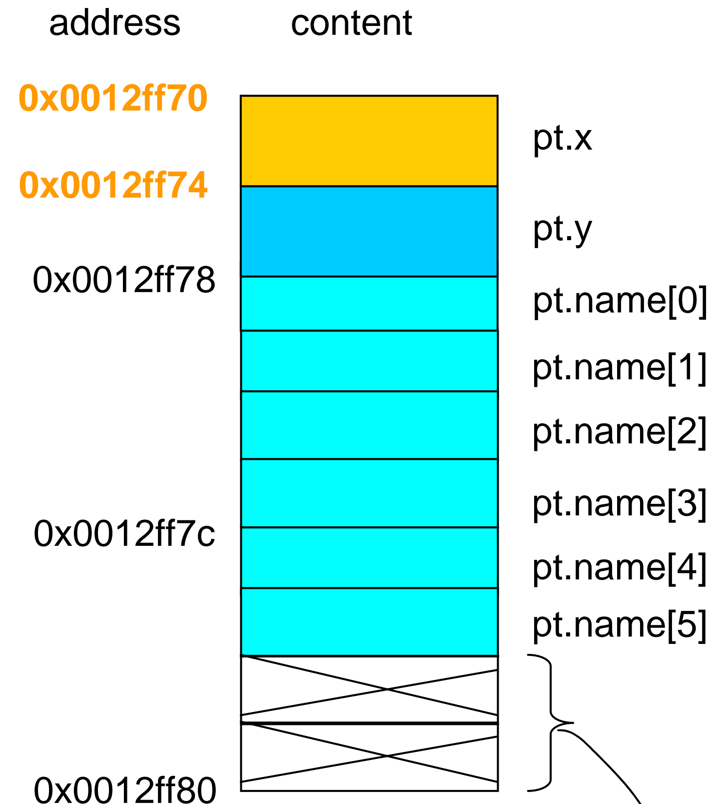
main(
mainC
KERNE

Name	Value
⊞ &pt	0x0012ff70
⊞ &pt.x	0x0012ff70
⊞ &pt.y	0x0012ff74
⊞ &pt.name	0x0012ff78
⊞ &maxpt	0x0012ff60
⊞ &maxpt.x	0x0012ff60
⊞ &maxpt.y	0x0012ff64
⊞ &maxpt.name	0x0012ff68

```
"F:\COURSE\2008SUMMER\C_LANG\
pt = (4, 3, Venus )
maxpt = (20, 30, Earth )
size of structure point = 16
Press any key to continue_
```

size of structure point

!= 14 (4+4+6)



Two bytes Padding by compiler

Padding and Alignment of structure [2]

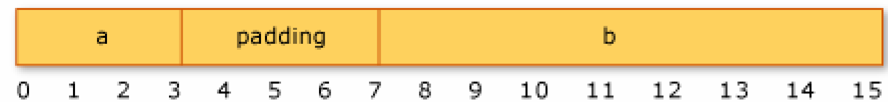
- The padding and alignment of members of structures and whether a bit field can straddle a storage-unit boundary.
- Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest
- Every data object has an alignment-requirement. For structures, the alignment-requirement is the largest alignment-requirement of its members. Every object is allocated an offset so that
  offset **%** *alignment-requirement* **==** 0
- When you use the /Zp[*n*] option, where *n* is 1, 2, 4, 8, or 16, each structure member after the first is stored on byte boundaries that are either the alignment requirement of the field or the packing size (*n*), default is 4.

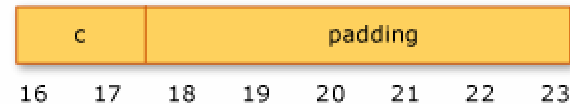
Padding and Alignment of structure [3]

```c
#include <stdio.h>

// word = 2 bytes, doubleword = 4 bytes, quadword = 8 bytes
struct S1 {
    short a ;  // size = 2 bytes, alignment = 2 bytes;
} ;

struct S2 {  // size = 24 bytes, alignment = quadword
    int    a ;
    double b ;
    short c ;
} ;

struct S3 {  // size = 12 bytes, alignment = doublewor…
    char   a ;
    short b ;
    char   c ;
    int    d ;
} ;

int main( int argc, char* argv[] )
{
    struct S1  x ;
    struct S2  y ;
    struct S3  z ;
    printf("size of struct S1 = %d\n", sizeof(struct S1) ) ;
    printf("size of struct S1 = %d\n", sizeof(struct S2) ) ;
    printf("size of struct S1 = %d\n", sizeof(struct S3) ) ;
    return 0 ;
}
```

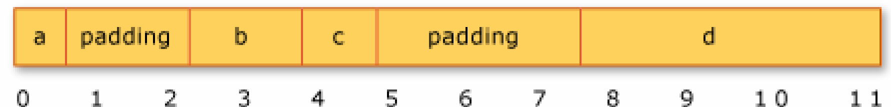
Structure S1

Structure S2





Structure S3



Example from MSDN Library

Padding and Alignment of structure [4]

suggested alignment for the scalar members of unions and structures

from MSDN Library

Scalar Type	C Data Type	Required Alignment
INT8	**char**	Byte
UINT8	**unsigned char**	Byte
INT16	**short**	Word
UINT16	**unsigned short**	Word
INT32	**int, long**	Doubleword
UINT32	**unsigned int, unsigned long**	Doubleword
INT64	**__int64**	Quadword
UINT64	**unsigned __int64**	Quadword
FP32 (single precision)	**float**	Doubleword
FP64 (double precision)	**double**	Quadword
POINTER	*	Quadword

Padding and Alignment of structure [5]

alignment rules

- The alignment of an array is the same as the alignment of one of the elements of the array.
- The alignment of the beginning of a structure is the maximum alignment of any individual member. Each member within the structure must be placed at its proper alignment as defined in the previous table, which may require implicit internal padding, depending on the previous member.
- Structure size must be an integral multiple of its alignment.
- It is possible to align data in such a way as to be greater than the alignment requirements as long as the previous rules are maintained.
- An individual compiler may adjust the packing of a structure for size reasons.

pointer to structure

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

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
    char name[6] ; // name of the point
} ;

int main( int argc, char* argv[] )
{
    struct point  *pt = NULL ; // pt is a pointer

    pt = (struct point *) malloc( sizeof(struct point) ) ;
    assert( pt ) ;

    pt->x = 4 ; // set x component of point pt as 4
    pt->y = 3 ; // set y component of point pt as 3
    strcpy( pt->name, "Venus") ; // set name of pt

    printf("*pt = (%d, %d, %s )\n", pt->x , pt->y, pt->name );

    printf("*pt = (%d, %d, %s )\n", (*pt).x , (*pt).y, (*pt).name );

    return 0 ;
}
```

1. ***pt->x*** is equivalent to ***(*p).x***

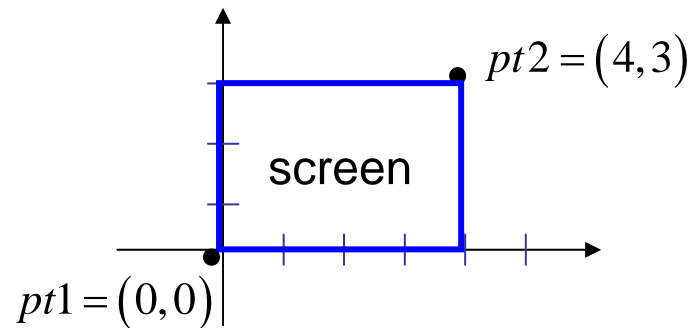
2. ****p.x*** is equivalent to ****(p.x)*** since dot operator has higher precedence than dereference operator

Precedence and Associativity of C Operators

Symbol1	Type of Operation	Associativity
[ ] ( ) . –> postfix ++ and postfix ––	Expression	Left to right
prefix ++ and prefix –– **sizeof &** * + – ~ !	Unary	Right to left

Nested structure

```c
#include <stdio.h>

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

struct rect {
    struct point pt1 ; // bottom-left point
    struct point pt2 ; // top-right point
} ;

int main( int argc, char* argv[] )
{
    struct rect  screen ;

    screen.pt1.x = 0 ;
    screen.pt1.y = 0 ;

    screen.pt2.x = 4 ;
    screen.pt2.y = 3 ;

    printf("screen = (%d, %d), (%d, %d)\n",
        screen.pt1.x , screen.pt1.y,
        screen.pt2.x , screen.pt2.y   );

    return 0 ;
}
```



screen.pt1.x is equivalent to (**screen.pt1**)*.x*

Since dot operator has ***left-right*** assciativity

and **screen.pt1** is alos a (point) structure

Precedence and Associativity of C Operators

Symbol1	Type of Operation	Associativity
[ ] ( ) . –> postfix ++ and postfix ––	Expression	Left to right
prefix ++ and prefix –– **sizeof & * + – ~ !**	Unary	Right to left

OutLine

- Basics of structures
- Structures and functions
- Arrays of structures
- Self-referential structure

Function returns structure [1]

```c
#include <stdio.h>

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

struct rect {
    struct point pt1 ; // bottom-left point
    struct point pt2 ; // top-right point
} ;

struct point makePoint(int x, int y) ;

int main( int argc, char* argv[] )
{
    struct rect  screen ;

    screen.pt1 = makePoint(0, 0) ;

    screen.pt2 = makePoint(4, 3) ;

    printf("screen = (%d, %d), (%d, %d)\n",
        screen.pt1.x , screen.pt1.y,
        screen.pt2.x , screen.pt2.y   );

    return 0 ;
}
```

1. declare function ***makePoint*** which accepts two integer and return a structure

3. assign return-value (structure ***temp***) to structure ***screen.pt1.*** Such assignment is done by compiler, it does memory copy.

```c
struct point makePoint(int x, int y)
{
    struct point temp ;
    temp.x = x ;
    temp.y = y ;
    return temp ;
}
```

2. structure **temp** is a local variable, ***x*** of **temp.x** is a field name but ***x*** itself is also a local variable, both x’s have different meanings.

Function returns structure [2]

```c
struct rect {
    struct point pt1 ; // bottom-left point
    struct point pt2 ; // top-right point
} ;

struct point makePoint(int x, int y) ;

int main( int argc, char* argv[] )
{
    struct rect  screen ;

    screen.pt1 = makePoint(0, 0) ;

    screen.pt2 = makePoint(4, 3) ;

    printf("screen = (%d, %d), (%d, %d)\n",
        screen.pt1.x , screen.pt1.y,
        screen.pt2.x , screen.pt2.y );
```

按 F11 進入 makePoint

Context: main(int, char * *)

Name	Value
argc	1
⊞ argv	0x003720c0
⊞ screen	{...}
⊞ screen.pt1	{...}

main(i
mainCR
KERNEL

Name	Value
⊟ &screen	0x0012ff70
⊟ pt1	{...}
x	-858993460
y	-858993460
⊟ pt2	{...}
x	-858993460
y	-858993460

Function returns structure [3]

```
struct point makePoint(int x, int y)
{
    struct point temp ;
    temp.x = x ;
    temp.y = y ;
    return temp ;
}
```

按 F10 二次, assign x and y to temp

Context: makePoint(int, int)

Name	Value
⊞ temp	{...}
temp.x	-858993460
x	0
y	0

makePo
main(i
mainCR
KERNEL

Name	Value
&screen	CXX0017: Error: symbol "screen" not found
⊟ &temp	0x0012fefc
x	-858993460
y	-858993460
⊞ &x	0x0012ff0c
⊞ &y	0x0012ff10

```
struct point makePoint(int x, int y)
{
    struct point temp ;
    temp.x = x ;
    temp.y = y ;
    return temp ;
}
```

按 F10 離開 makePoint

Context: makePoint(int, int)

Name	Value
⊞ temp	{...}
temp.y	0
y	0

makePo
main(i
mainCR
KERNEL

Name	Value
&screen	CXX0017: Error: symbol "screen" not found
⊟ &temp	0x0012fefc
x	0
y	0
⊞ &x	0x0012ff0c
⊞ &y	0x0012ff10

Function returns structure [4]

```
screen.pt1 = makePoint(0, 0) ;
screen.pt2 = makePoint(4, 3) ;
```

按 F10 作 copy 動作

Context: main(int, char * *)

Name	Value
argc	1
argv	0x003720c0
screen	{...}
screen.pt1	{...}
makePoint ret	{...}

main(i
mainCR
KERNEL

Name	Value
&screen	0x0012ff70
pt1	{...}
x	-858993460
y	-858993460
pt2	{...}
x	-858993460
y	-858993460

```
screen.pt1 = makePoint(0, 0) ;
screen.pt2 = makePoint(4, 3) ;
```

Context: main(int, char * *)

Name	Value
screen	{...}
screen.pt1	{...}
screen.pt2	{...}

main(i
mainCR
KERNEL

Name	Value
&screen	0x0012ff70
pt1	{...}
x	0
y	0
pt2	{...}
x	-858993460
y	-858993460

update ***screen.pt1***

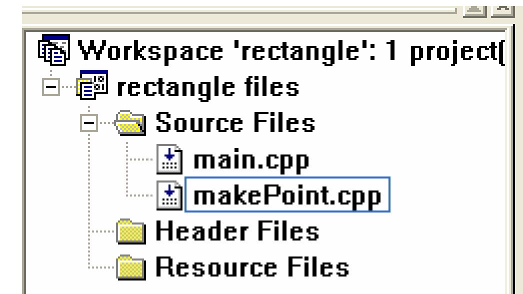
Header file (標頭檔) [1]

main.cpp

```
#include <stdio.h>

struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

struct rect {
    struct point pt1 ; // bottom-left point
    struct point pt2 ; // top-right point
} ;

struct point makePoint(int x, int y) ;

int main( int argc, char* argv[] )
{
    struct rect  screen ;

    screen.pt1 = makePoint(0, 0) ;

    screen.pt2 = makePoint(4, 3) ;

    printf("screen = (%d, %d), (%d, %d)\n",
        screen.pt1.x , screen.pt1.y,
        screen.pt2.x , screen.pt2.y   );

    return 0 ;
}
```

makePoint.cpp

```
struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

struct rect {
    struct point pt1 ; // bottom-left point
    struct point pt2 ; // top-right point
} ;

struct point makePoint(int x, int y)
{
    struct point temp ;
    temp.x = x ;
    temp.y = y ;
    return temp ;
}
```



Question: can we eliminate duplication of structure definition?

Header file (標頭檔) [2]

New

Files | Projects | Other Documents

Active Server Page
Binary File
Bitmap File
C/C++ Header File
C++ Source File
Cursor File
Fortran Fixed Format Source File
Fortran Free Format Source File
HTML Page
Icon File

Add to project: rectangle

File: point_rect

Location: F:\COURSE\2008SUMMER\C_LA

1. 選擇 header file

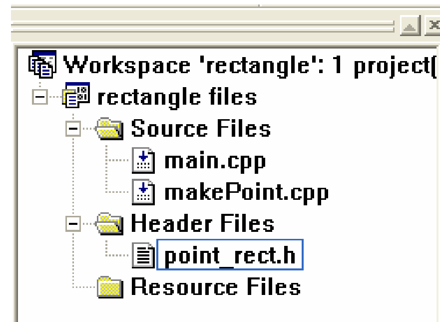
2. 檔名 point_rect.h

main.cpp

```
#include <stdio.h>

#include "point_rect.h"

struct point makePoint(int x, int y) ;

int main( int argc, char* argv[] )
{
    struct rect  screen ;

    screen.pt1 = makePoint(0, 0) ;

    screen.pt2 = makePoint(4, 3) ;

    printf("screen = (%d, %d), (%d, %d)\n",
        screen.pt1.x , screen.pt1.y,
        screen.pt2.x , screen.pt2.y   );

    return 0 ;
}
```

makePoint.cpp

```
#include "point_rect.h"

struct point makePoint(int x, int y)
{
    struct point temp ;
    temp.x = x ;
    temp.y = y ;
    return temp ;
}
```



point_rect.h

```
struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} ;

struct rect {
    struct point pt1 ; // bottom-left point
    struct point pt2 ; // top-right point
} ;
```

Header file (標頭檔): typedef [3]

point_rect.h

```
typedef struct point {
    int x ;  // x component of a point
    int y ;  // y component of a point
} pointType ;


typedef struct rect {
    struct point pt1 ; // bottom-left point
    struct point pt2 ; // top-right point
} rectType ;
```

symbol ***pointType*** is equivalent to ***struct point***

makePoint.cpp

```
#include "point_rect.h"

pointType makePoint(int x, int y)
{
    pointType temp ;

    temp.x = x ;
    temp.y = y ;

    return temp ;
}
```

main.cpp

```
#include <stdio.h>

#include "point_rect.h"

pointType makePoint(int x, int y) ;

int main( int argc, char* argv[] )
{
    rectType  screen ;

    screen.pt1 = makePoint(0, 0) ;

    screen.pt2 = makePoint(4, 3) ;

    printf("screen = (%d, %d), (%d, %d)\n",
        screen.pt1.x , screen.pt1.y,
        screen.pt2.x , screen.pt2.y   );

    return 0 ;
}
```

OutLine

- Basics of structures
- Structures and functions
- Arrays of structures
  - initialization
  - linear search and binary search
- Self-referential structure

Array of structures [1]

```c
#include <stdio.h>

#include "key.h"

// list of all C keywords, see page 192 of textbook
keyType  keytab[] = {
    {"auto"    ,0}, {"double",0}, {"int"     ,0}, {"struct"  ,0},
    {"break"   ,0}, {"else"  ,0}, {"long"    ,0}, {"switch"  ,0},
    {"case"    ,0}, {"enum"  ,0}, {"register",0}, {"typedef" ,0},
    {"char"    ,0}, {"extern",0}, {"return"  ,0}, {"union"   ,0},
    {"const"   ,0}, {"float" ,0}, {"short"   ,0}, {"unsigned",0},
    {"continue",0}, {"for"   ,0}, {"signed"  ,0}, {"void"    ,0},
    {"default" ,0}, {"goto"  ,0}, {"sizeof"  ,0}, {"volatile",0},
    {"do"      ,0}, {"if"    ,0}, {"static"  ,0}, {"while"   ,0}
} ;

// quantity NKEYS is number of keywords in array keytab
#define NKEYS  ( sizeof keytab / sizeof(keyType) )

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

    for (i=0 ; i < NKEYS ; i++){
        if ( 0 == i % 3 ) printf("\n");
        printf("keytab[%2d] = %6s\t", i, keytab[i].word );
    }
    printf("\n");

    return 0 ;
}
```

1. number of elements in array ***keytab*** is determined by compiler

2. Since compiler know number of elements in array ***keytab***, hence **NKEYS** can be determined by coompiler

3. ***keytab[i].word*** is equivalent to ***(keytab[i]).word***

since ***[ ]*** and **.** Have the same precedence and associativity is left-right

```c
/* write a program to count the occurrences
   of each C keyword.
   NKEYS = number of C keyword
   char* keyword[NKEYS] : record name of keyword
   int   keycount[NKEYS] : record occurrence
*/

typedef struct key {
    char *word ; // keyword of C-language
    int  count ; // number of keyword in a file
} keyType ;
```

Array of structures [2]

```
int main( int argc, char* argv[] )
{
    int i ;

    for (i=0 ; i < NKEYS ; i++){
        if ( 0 == i % 3 ) printf("\n");
        printf("keytab[%2d] = %6s\t", i, keytab[i].word );
    }
    printf("\n");

    return 0 ;
}
```

Context: main(in

Name	Value
argc	1
⊞ argv	0x00372
i	-858993
⊞ keytab	0x00424 struct key *

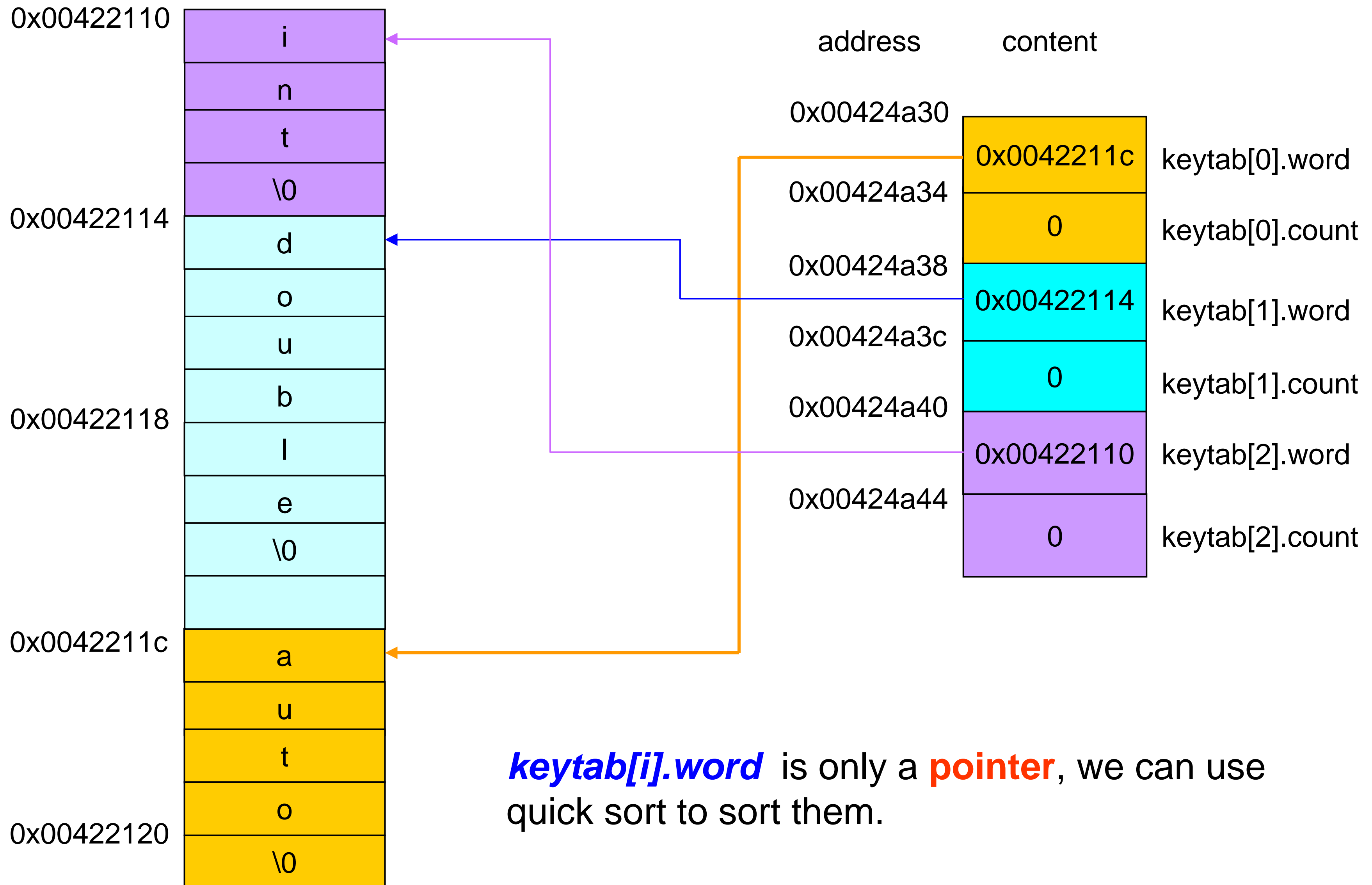
Name	Value
⊞ keytab	0x00424a30 struct key * keytab
⊟ &keytab[0]	0x00424a30 struct key * keytab
⊞ word	0x0042211c "auto"
count	0
⊟ &keytab[1]	0x00424a38
⊞ word	0x00422114 "double"
count	0
⊟ &keytab[2]	0x00424a40
⊞ word	0x00422110 "int"
count	0

address	content	
0x00424a30	0x0042211c	keytab[0].word
0x00424a34	0	keytab[0].count
0x00424a38	0x00422114	keytab[1].word
0x00424a3c	0	keytab[1].count
0x00424a40	0x00422110	keytab[2].word
0x00424a44	0	keytab[2].count

keytab is unsorted array under lexicographic order

```
keytab[ 0] =   auto     keytab[ 1] = double     keytab[ 2] =    int
keytab[ 3] = struct     keytab[ 4] =  break     keytab[ 5] =   else
keytab[ 6] =   long     keytab[ 7] = switch     keytab[ 8] =   case
keytab[ 9] =   enum     keytab[10] = register   keytab[11] = typedef
keytab[12] =   char     keytab[13] = extern     keytab[14] = return
keytab[15] =  union     keytab[16] =  const     keytab[17] =  float
keytab[18] =  short     keytab[19] = unsigned   keytab[20] = continue
keytab[21] =    for     keytab[22] = signed     keytab[23] =   void
keytab[24] = default    keytab[25] =   goto     keytab[26] = sizeof
keytab[27] = volatile   keytab[28] =     do     keytab[29] =     if
keytab[30] = static     keytab[31] =  while
Press any key to continue
```

Array of structures [3]



keytab[i].word is only a **pointer**, we can use quick sort to sort them.

Array of structures [4]

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

#include "key.h"

// list of all C keywords, see page 192 of textbook
keyType  keytab[] = {
    {"auto"    ,0}, {"double",0}, {"int"     ,0}, {"struct"  ,0},
    {"break"   ,0}, {"else"  ,0}, {"long"    ,0}, {"switch"  ,0},
    {"case"    ,0}, {"enum"  ,0}, {"register",0}, {"typedef" ,0},
    {"char"    ,0}, {"extern",0}, {"return"  ,0}, {"union"   ,0},
    {"const"   ,0}, {"float" ,0}, {"short"   ,0}, {"unsigned",0},
    {"continue",0}, {"for"   ,0}, {"signed"  ,0}, {"void"    ,0},
    {"default" ,0}, {"goto"  ,0}, {"sizeof"  ,0}, {"volatile",0},
    {"do"      ,0}, {"if"    ,0}, {"static"  ,0}, {"while"   ,0}
} ;

// quantity NKEYS is number of keywords in array keytab
#define NKEYS  ( sizeof keytab / sizeof(keyType) )

int  keyword_cmp( keyType *s, keyType *t )
{
    return strcmp( s->word, t->word ) ;
}


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

    qsort( (void*) keytab, (size_t) NKEYS, (size_t) sizeof(keyType),
         (int (*)(const void*, const void*)) &keyword_cmp ) ;

    for (i=0 ; i < NKEYS ; i++){
        if ( 0 == i % 3 ) printf("\n");
        printf("keytab[%d] = %6s\t", i, keytab[i].word );
    }
    printf("\n");

    return 0 ;
}
```

sorted ***keytab***

```
keytab[0] =   auto      keytab[1] =  break      keytab[2] =   case
keytab[3] =   char      keytab[4] =  const      keytab[5] = continue
keytab[6] = default     keytab[7] =     do      keytab[8] = double
keytab[9] =   else      keytab[10] =   enum     keytab[11] = extern
keytab[12] =  float     keytab[13] =    for     keytab[14] =   goto
keytab[15] =     if     keytab[16] =    int     keytab[17] =   long
keytab[18] = register   keytab[19] = return     keytab[20] =  short
keytab[21] = signed     keytab[22] = sizeof     keytab[23] = static
keytab[24] = struct     keytab[25] = switch     keytab[26] = typedef
keytab[27] =  union     keytab[28] = unsigned   keytab[29] =   void
keytab[30] = volatile   keytab[31] =  while
Press any key to continue
```

Array of structures: linear search [5]

main.cpp

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

#include "key.h"

// list of all C keywords, see page 192 of textbook
keyType  keytab[] = {
    {"auto"    ,0}, {"double",0}, {"int"     ,0}, {"struct"  ,0},
    {"break"   ,0}, {"else"  ,0}, {"long"    ,0}, {"switch"  ,0},
    {"case"    ,0}, {"enum"  ,0}, {"register",0}, {"typedef" ,0},
    {"char"    ,0}, {"extern",0}, {"return"  ,0}, {"union"   ,0},
    {"const"   ,0}, {"float" ,0}, {"short"   ,0}, {"unsigned",0},
    {"continue",0}, {"for"   ,0}, {"signed"  ,0}, {"void"    ,0},
    {"default" ,0}, {"goto"  ,0}, {"sizeof"  ,0}, {"volatile",0},
    {"do"      ,0}, {"if"    ,0}, {"static"  ,0}, {"while"   ,0}
} ;

// quantity NKEYS is number of keywords in array keytab
#define NKEYS  ( sizeof keytab / sizeof(keyType) )

int  keyword_cmp( keyType *s, keyType *t )
{
    return strcmp( s->word, t->word ) ;
}

int linear_search( char *key, keyType *base, size_t n ) ;

int main( int argc, char* argv[] )
{
    char key[] = "endfor" ;

    qsort( (void*) keytab, (size_t) NKEYS, (size_t) sizeof(keyType),
         (int (*)(const void*, const void*)) &keyword_cmp ) ;

    if ( 0 > linear_search( key, keytab, NKEYS) ){
        printf(" \"%s\" is not a keyword\n", key);
    }else{
        printf(" \"%s\" is a keyword\n", key );
    }

    return 0 ;
}
```

linear_search.cpp

```
#include <stddef.h>
#include <string.h>
#include "key.h"

/*
  Given keyType array base[0], ... base[n-1]
  check if key is a keyword in array base
 */
int linear_search( char *key, keyType *base, size_t n )
{
    size_t i ;

    for( i=0 ; i < n ; i++ ){
        if ( 0 == strcmp( key, base[i].word ) )
            return i ;
    }
    return -1 ; // not found
}
```

$$protocol(協定):\ \text{strcmp}(s,t)\ \text{return} \begin{cases} <0 & \text{if } s<t \\ =0 & \text{if } s=t \\ >0 & \text{if } s>t \end{cases}$$

```
"F:\COURSE\2008SUMMER\C_LA
 "endfor" is not a keyword
Press any key to continue_
```

Observation of linear search

- Data type of ***key*** and ***base*** are immaterial, we only need to provide comparison operator. In other words, framework of linear search is independent of comparison operation.
- We have two choices for "return location of ***base[j]***", one is array index and the other is address of ***base[j]*** , which one is better?

pseudocode

Given array $base[0:n-1]$ and a search key

key and $base$ may have different data type

$for\ \ j = 0:1:n-1$

  if $base[j] == key$ then

    return location of $base[j]$

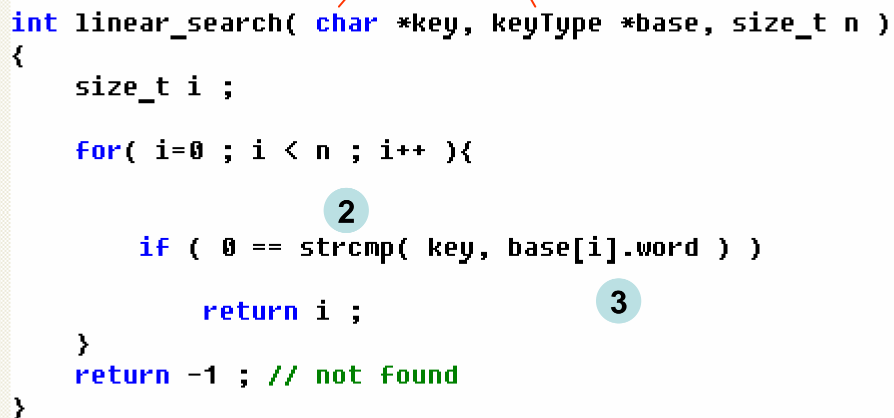
$endfor$

return not-found

drawback of current version

1. Explicitly specify type of ***key*** and type of ***base***, this violates observation “data type of ***key*** and ***base*** are immaterial”

2. Explicitly specify comparison operation ***strcmp***, this violates “comparison operator is independent of linear search”



3. Specify ***base[i].word***, word is a field binding to data type ***keyType***, this violates “data type of ***key*** and ***base*** are immaterial”.

Besides, ***base[i]*** require data type of base implicitly since compiler needs to translate address of ***base[i]*** as ***base+sizeof(keyType)*i*** , this violates “data type of ***key*** and ***base*** are immaterial”.

framework of linear search [1]

```c
#include <stddef.h>
#include <string.h>
#include "key.h"

/* Given keyType array base[0], ... base[n-1]
   check if key is a keyword in array base */

void*   linear_search( const void *key, const void *base,
        size_t n, size_t size,
        int (*cmp)(const void *keyval, const void *atum)
        )
{
    size_t i ;
    char *a_i ; // &base[i]
    char *a = (char*) base ;

    for( i=0 ; i < n ; i++ ){
        a_i = a + size*i ;

        if ( 0 == (*cmp)( key, a_i ) ){
            return a_i ;
        }
    }
    return NULL ; // not found
}
```

```c
int linear_search( char *key, keyType *base, size_t n )
{
    size_t i ;

    for( i=0 ; i < n ; i++ ){

        if ( 0 == strcmp( key, base[i].word ) )

            return i ;
    }
    return -1 ; // not found
}
```

1. **NOT** explicitly specify type of ***key*** and type of ***base***

2. **NOT** explicitly specify comparison operation ***strcmp***

3. **NOT** specify ***base[i].word,*** also replace ***&base[i]*** by ***base+sizeof(keyType)*i***

Question: why do we need character pointer ***a***? Can we use ***base*** directly?

framework of linear search [2]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "key.h"
// list of all C keywords, see page 192 of textbook
keyType  keytab[] = {
    {"auto"    ,0}, {"double",0}, {"int"     ,0}, {"struct"  ,0},
    {"break"   ,0}, {"else"  ,0}, {"long"    ,0}, {"switch"  ,0},
    {"case"    ,0}, {"enum"  ,0}, {"register",0}, {"typedef" ,0},
    {"char"    ,0}, {"extern",0}, {"return"  ,0}, {"union"   ,0},
    {"const"   ,0}, {"float" ,0}, {"short"   ,0}, {"unsigned",0},
    {"continue",0}, {"for"   ,0}, {"signed"  ,0}, {"void"    ,0},
    {"default" ,0}, {"goto"  ,0}, {"sizeof"  ,0}, {"volatile",0},
    {"do"      ,0}, {"if"    ,0}, {"static"  ,0}, {"while"   ,0}
} ;
// quantity NKEYS is number of keywords in array keytab
#define NKEYS  ( sizeof keytab / sizeof(keyType) )

int  keyword_cmp( char *keyval, keyType *datum )
{
    return strcmp( keyval, datum->word ) ;
}

void*   linear_search( const void *key, const void *base,
        size_t n, size_t size,
        int (*cmp)(const void *keyval, const void *atum)  ) ;

int main( int argc, char* argv[] )
{
    char key[] = "endfor" ;
    keyType   *found_key ; // result of linear search

    qsort( (void*) keytab, (size_t) NKEYS, (size_t) sizeof(keyType),
         (int (*)(const void*, const void*)) &keyword_cmp ) ;

    found_key = (keyType*)   linear_search( key, keytab,
                               NKEYS, sizeof(keyType),
            (int (*)(const void*, const void*)) &keyword_cmp ) ;

    if ( NULL ==  found_key ){
        printf(" \"%s\" is not a keyword\n", key);
    }else{
        printf(" \"%s\" is a keyword\n", found_key->word );
    }

    return 0 ;
}
```

1. search ***key*** must be consistent with ***keyval*** in comparison operator, say ***key*** and ***keyval*** have the same data type, pointer to content of search key

2. ***keytab[i]*** must be consistent with **found_key*, they must be the same type and such type has ***sizeof(keyType)*** bytes

```
"F:\COURSE\2008SUMMER\C_LA
 "endfor" is not a keyword
Press any key to continue
```

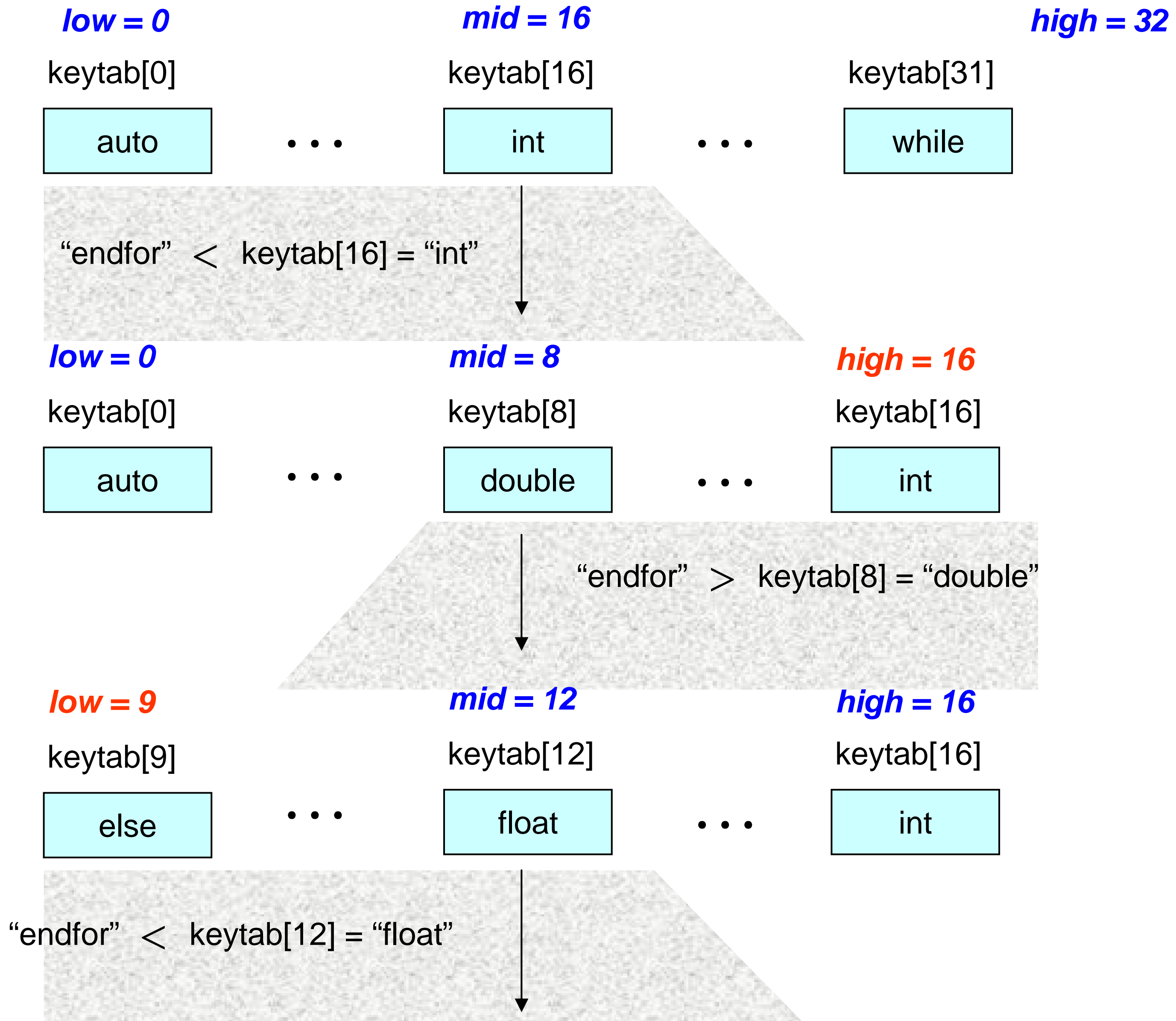
framework of binary search [1]

```c
#include <stddef.h>

/* Given keyType array base[0], ... base[n-1]
   check if key is a keyword in array base */

void*   binsearch( const void *key, const void *base,
        size_t n, size_t size,
        int (*cmp)(const void *keyval, const void *datum)
        )
{
    size_t  low, high, mid ; // index of array base,
             // always keep low < mid < high
    int cond ; // comparison result of key and base[i]
    char *a_i ; // &base[i]
    char *a = (char*) base ;

    low = 0 ; high = n ;
    while( low < high ){
        mid = low + (high - low)/2 ;
        a_i = a + size*mid ;
        cond = (*cmp)( key, a_i ) ;
        if ( 0 > cond )
            high = mid ;
        else if ( 0 < cond )
            low = mid + 1 ;
        else
            return a_i ;
    }
    return NULL ; // not found
}
```

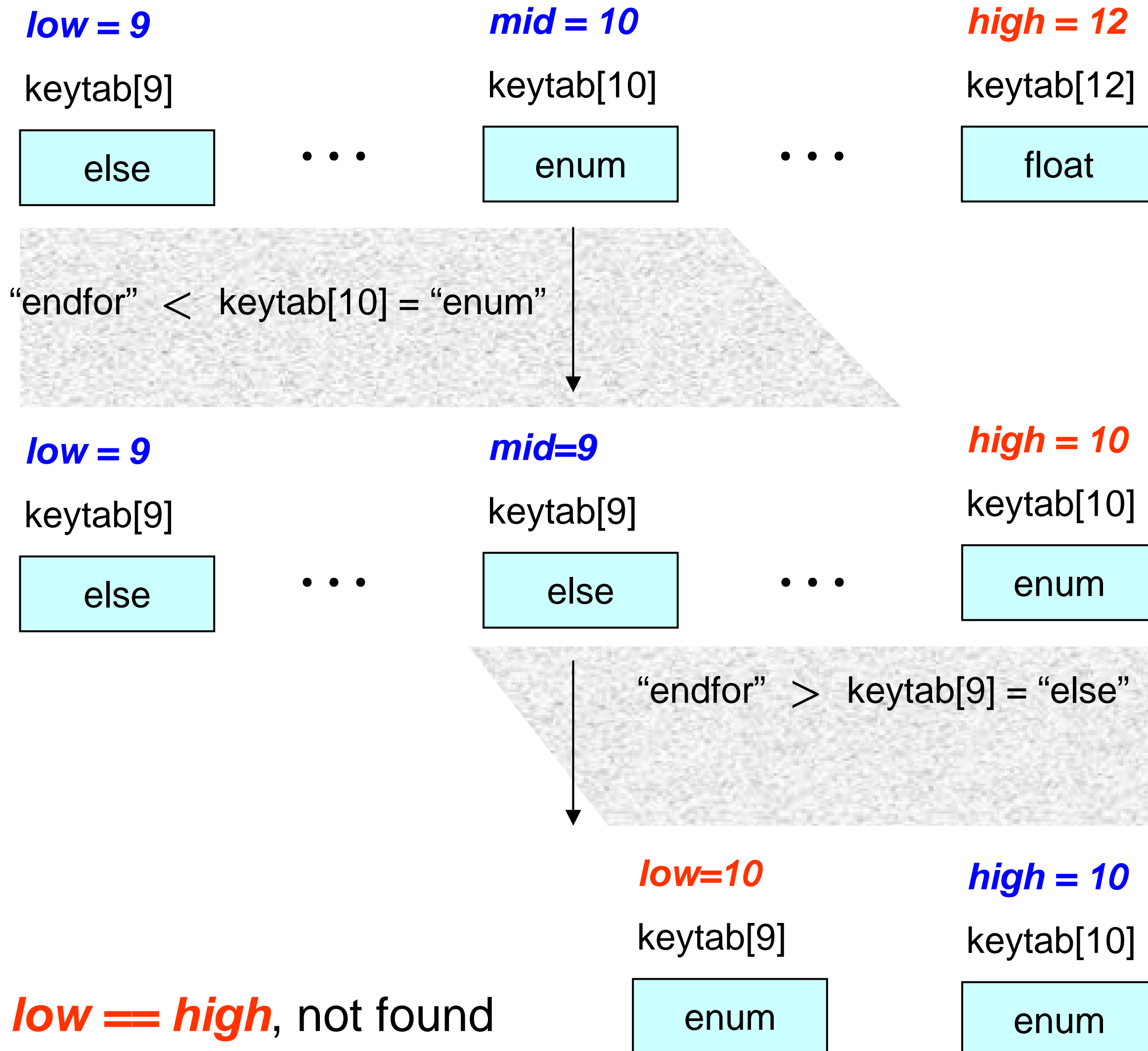
```c
#include <stddef.h>

/* Given keyType array base[0], ... base[n-1]
   check if key is a keyword in array base */

void*  linear_search( const void *key, const void *base,
        size_t n, size_t size,
    int (*cmp)(const void *keyval, const void *atum)
        )
{
    size_t i ;
    char *a_i ; // &base[i]
    char *a = (char*) base ;

    for( i=0 ; i < n ; i++ ){
        a_i = a + size*i ;

        if ( 0 == (*cmp)( key, a_i ) ){
            return a_i ;
        }
    }
    return NULL ; // not found
}
```

since “endfor” is not a keyword, under linear search algorithm, we need to compare all keywords to reject “endfor”. We need another efficient algorithm, binary search, which is the best.

framework of binary search [2]



framework of binary search [3]



framework of binary search : standard library [4]

Page 253 in textbook, binary search algorithm is included in stadard C library, **stdlib.h**

```
void*  bsearch( const void *key, const void *base,
              size_t n, size_t size,
           int (*cmp)(const void *keyval, const void *datum)   )
```

Objective : *bsearch* seraches *base[0],…, base[n-1]* for an item that matches **key*

Requirement : ***base*** must be in ascending order

$$protocol\left(\text{協定}\right):\ *\mathrm{cmp}\left(s,t\right)\ \text{return}\ \begin{cases} <0 & \text{if } s<t \\ =0 & \text{if } s=t \\ >0 & \text{if } s>t \end{cases}$$

Return : pointer to a matching item or *NULL* if none exists

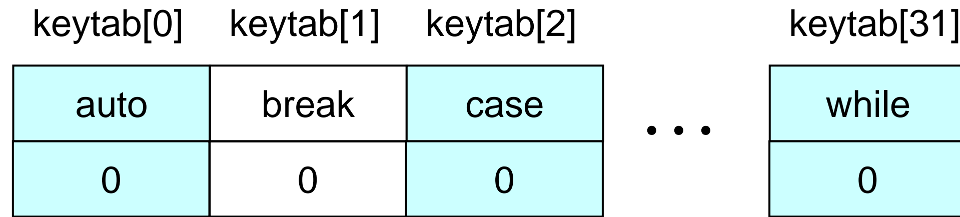
OutLine

- Basics of structures
- Structures and functions
- Arrays of structures
- Self-referential structure (linked list)
  - formulation
  - traversal
  - de-allocation

Self-referential structure: linked list [1]

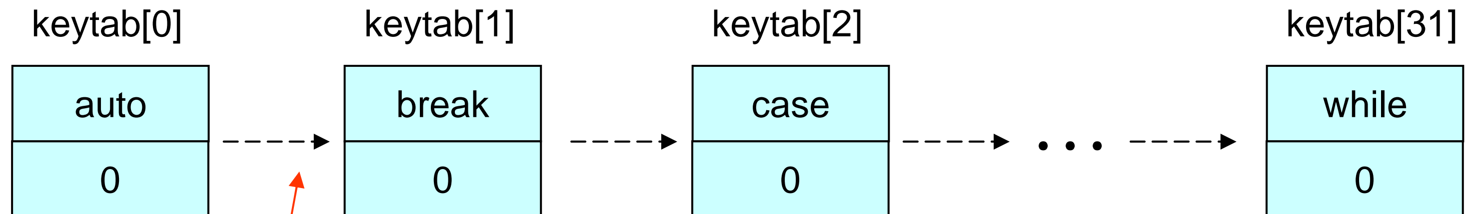
```c
typedef struct key {
    char *word ; // keyword of C-language
    int  count ; // number of keyword in a file
} keyType ;
```

contiguous (array)



dis-contiguous

(Linked list)



How to implement such logical link?

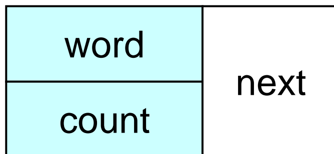
Self-referential structure: linked list [2]

keyList.h

```
typedef struct keyListEle {
    char word[16] ;          // keyword of C-language
    int  count ;             // number of keyword in a file
    struct keyListEle  *next ; // next entry in the chain
} keyListEleType ;
```

key.h

```
typedef struct key {
    char *word ; // keyword of C-language
    int  count ; // number of keyword in a file
} keyType ;
```



Discontiguous (Linked list)

auto 0 → break 0 → case 0 → char 0 → . . . → while 0 → null

Question: How to write code to implment this graph?

Self-referential structure: linked list [3]

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

#include "keyList.h"

int main( int argc, char* argv[] )
{
    keyListEleType  *keytabList = NULL ;
    keyListEleType  *unitEle = NULL ;
    keyListEleType  *elePtr = NULL ;

// first element in linked list
    unitEle = (keyListEleType*) malloc( sizeof(keyListEleType) ) ;
    assert( unitEle ) ;
    strcpy( unitEle->word, "auto" ) ;
    unitEle->count = 0 ;
    unitEle->next = NULL ;

    keytabList = unitEle ;
// second elements in linked list
    unitEle = (keyListEleType*) malloc( sizeof(keyListEleType) ) ;
    assert( unitEle ) ;
    strcpy( unitEle->word, "break" ) ;
    unitEle->count = 0 ;
    unitEle->next = NULL ;

    keytabList->next = unitEle ;

    for ( elePtr = keytabList ; NULL != elePtr ; elePtr = elePtr->next ){
        printf("[0x%p] : word = %8s, count = %d, next = 0x%p\n", elePtr,
            elePtr->word, elePtr->count, elePtr->next );
    }

    return 0 ;
}
```

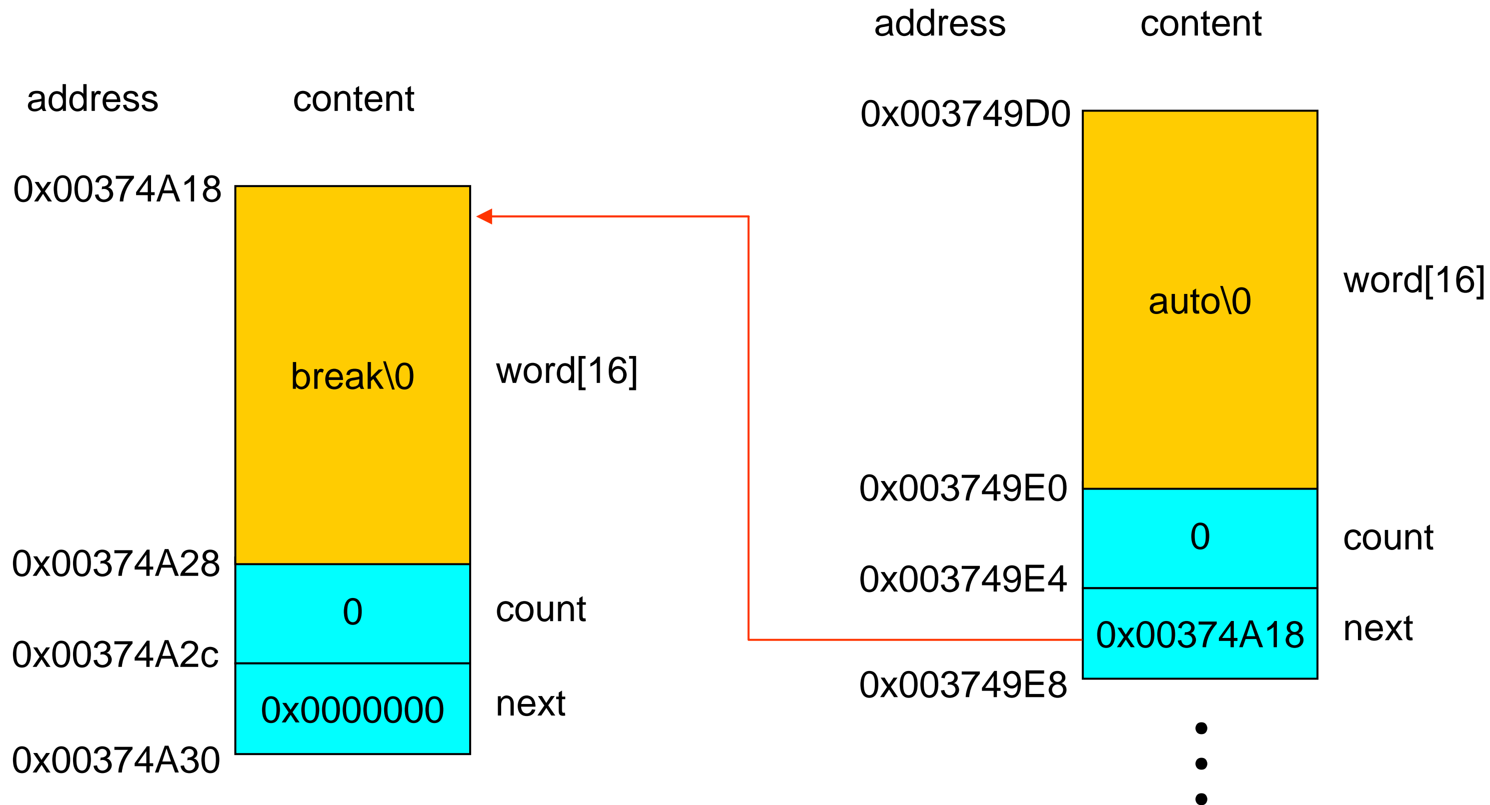
```
"F:\COURSE\2008SUMMER\C_LANG\EXAMPLE\CHAP6\linkList\Debug\linkList.e
[0x003749D0] : word =     auto, count = 0, next = 0x00374A18
[0x00374A18] : word =    break, count = 0, next = 0x00000000
Press any key to continue
```

1 — 1. create an element of keyword “auto” and set its address to ***keytabList***

2 — 2. create an element of keyword “break” and set it to be next element of ***keytabList***

3 — 3. sweep list pointed by ***keytabList***

Self-referential structure: linked list [4]




```
"F:\COURSE\2008SUMMER\C_LANG\EXAMPLE\CHAP6\linkList\Debug\linkList.e
[0x003749D0] : word =      auto, count = 0, next = 0x00374A18
[0x00374A18] : word =     break, count = 0, next = 0x00000000
Press any key to continue
```

Self-referential structure: linked list [5]

keytabList 0x00000000 empty list

---

unitEle 0x003749D0

```
// first element in linked list
    unitEle = (keyListEleType*) malloc( sizeof(keyListEleType) ) ;
    assert( unitEle ) ;
    strcpy( unitEle->word, "auto" ) ;
    unitEle->count = 0 ;
    unitEle->next = NULL ;
```

Address	Value	Field
0x003749D0	auto\0	word[16]
0x003749E0	0	count
0x003749E4	0x00000000	next

---

keytabList 0x003749D0

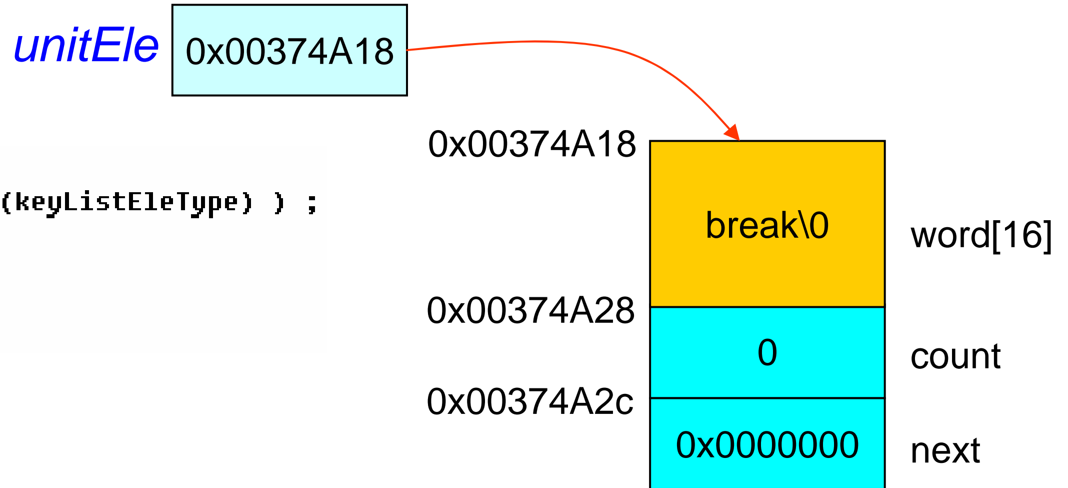
unitEle 0x003749D0

```
keytabList = unitEle ;
```

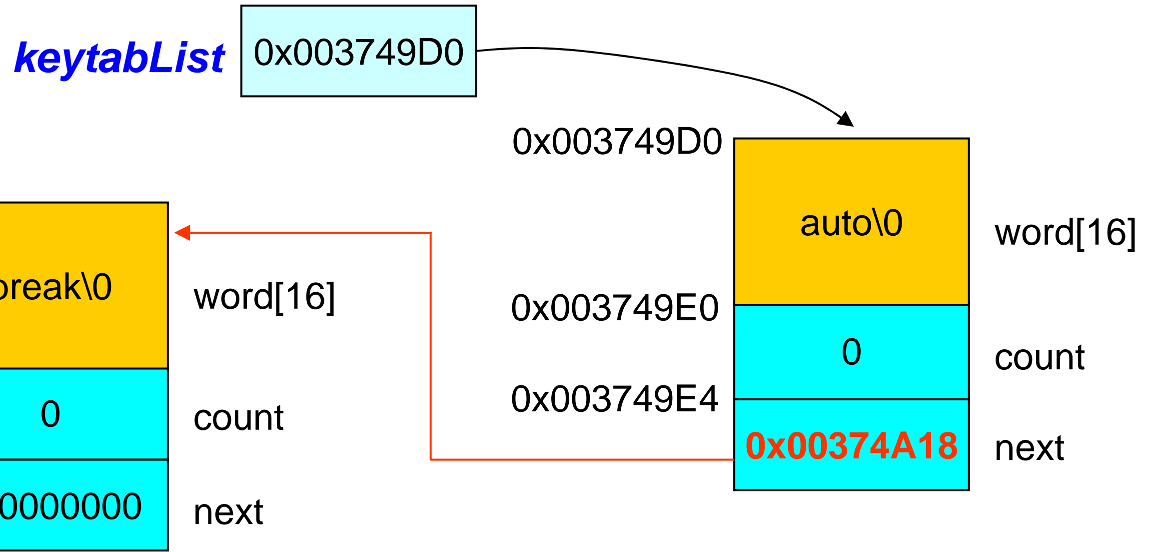
Address	Value	Field
0x003749D0	auto\0	word[16]
0x003749E0	0	count
0x003749E4	0x00000000	next

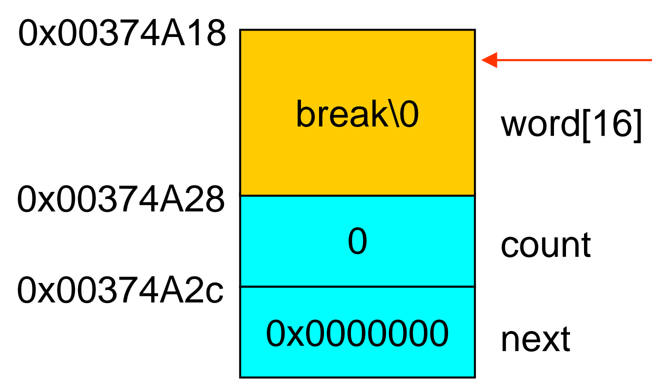
Self-referential structure: linked list [6]

```
// second elements in linked list
    unitEle = (keyListEleType*) malloc( sizeof(keyListEleType) ) ;
    assert( unitEle ) ;
    strcpy( unitEle->word, "break" ) ;
    unitEle->count = 0 ;
    unitEle->next = NULL ;
```



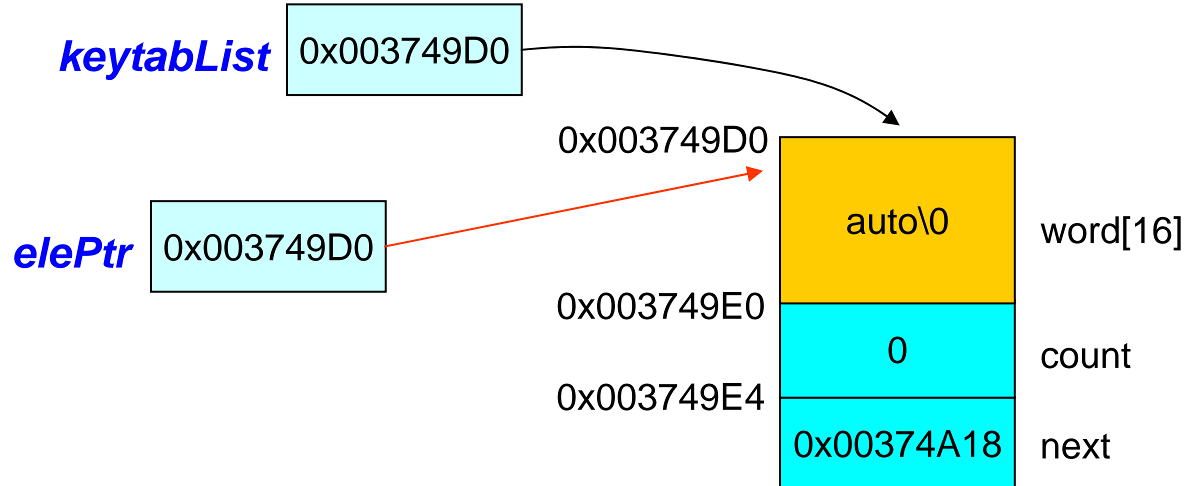

```
keytabList->next = unitEle ;
```





Self-referential structure: traverse linked list [7]

```
elePtr = keytabList
```




```
NULL != elePtr
```

elePtr = 0x003749D0 != 0

```
printf("[0x%p] : word = %8s, count = %d, next = 0x%p\n", elePtr,
    elePtr->word, elePtr->count, elePtr->next );
```

```
[0x003749D0] : word =     auto, count = 0, next = 0x00374A18
```

```
elePtr = elePtr->next
```

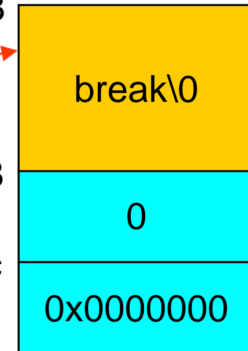
elePtr 0x00374A18

Self-referential structure: traverse linked list [8]

```
NULL != elePtr
```

elePtr 0x00374A18

elePtr = 0x00374A18 != 0




```
printf("[0x%p] : word = %8s, count = %d, next = 0x%p\n", elePtr,
    elePtr->word, elePtr->count, elePtr->next );
```

```
[0x00374A18] : word =    break, count = 0, next = 0x00000000
```

```
elePtr = elePtr->next
```

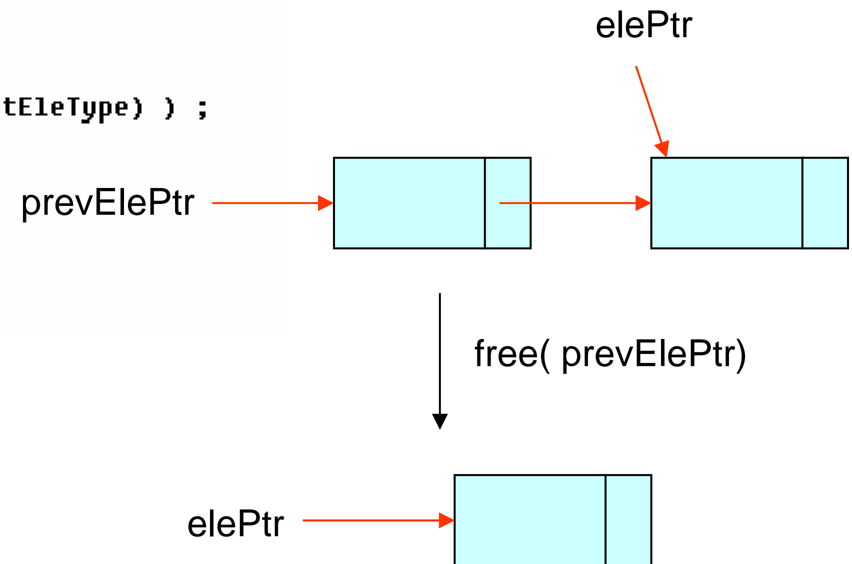
elePtr 0x00000000

```
NULL != elePtr
```

elePtr = 0x0000000 == 0, terminate

Self-referential structure: de-allocation [9]

```
int main( int argc, char* argv[] )
{
    keyListEleType  *keytabList = NULL ;
    keyListEleType  *unitEle = NULL ;
    keyListEleType  *elePtr = NULL ;
    keyListEleType  *prevElePtr = NULL ;

    printf("sizeof(keyListEleType) = %d\n", sizeof(keyListEleType) ) ;

// first element in linked list
    unitEle = (keyListEleType*) malloc( sizeof(keyListEleType) ) ;
    assert( unitEle ) ;
    strcpy( unitEle->word, "auto" ) ;
    unitEle->count = 0 ;
    unitEle->next = NULL ;

    keytabList = unitEle ;

// second elements in linked list
    unitEle = (keyListEleType*) malloc( sizeof(keyListEleType) ) ;
    assert( unitEle ) ;
    strcpy( unitEle->word, "break" ) ;
    unitEle->count = 0 ;
    unitEle->next = NULL ;

    keytabList->next = unitEle ;

// deallocate
    prevElePtr = keytabList ;
    while( NULL != prevElePtr ){
        elePtr = prevElePtr->next ;
        free( prevElePtr ) ;
        prevElePtr = elePtr ;
    }

    return 0 ;
}
```



Self-referential structure: wrong de-allocation [10]

```
// second elements in linked list
    unitEle = (keyListEleType*) malloc( sizeof(keyListEleType) ) ;
    assert( unitEle ) ;
    strcpy( unitEle->word, "break" ) ;
    unitEle->count = 0 ;
    unitEle->next = NULL ;

    keytabList->next = unitEle ;

    for ( elePtr = keytabList ; NULL != elePtr ;  elePtr = elePtr->next ){
        printf("[0x%p] : word = %8s, count = %d, next = 0x%p\n", elePtr,
            elePtr->word, elePtr->count, elePtr->next ) ;
    }

// wrong deallocation
    for ( elePtr = keytabList ; NULL != elePtr ;  elePtr = elePtr->next ){
         free( elePtr ) ;
    }
```