The C Programming Language – Part 4

(with material from Dr. Bin Ren, William & Mary Computer Science, and www.cpp.com)

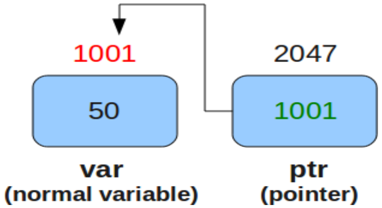
Overview

- Basic Concepts of Pointers
- Pointers and Arrays
- Pointers and Strings
- Dynamic Memory Allocation

Pointers

- **pointer – the address of something**

- **values of variables are stored in memory, at particular locations**
  - exact memory locations unknown at compile time
  - a location is identified and referenced with an address
    - analogous to identifying a house's location via an address



← address in memory

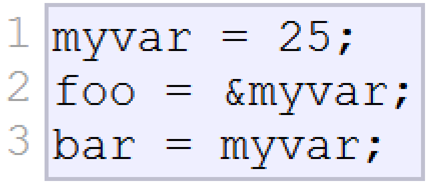
← Value

← variable

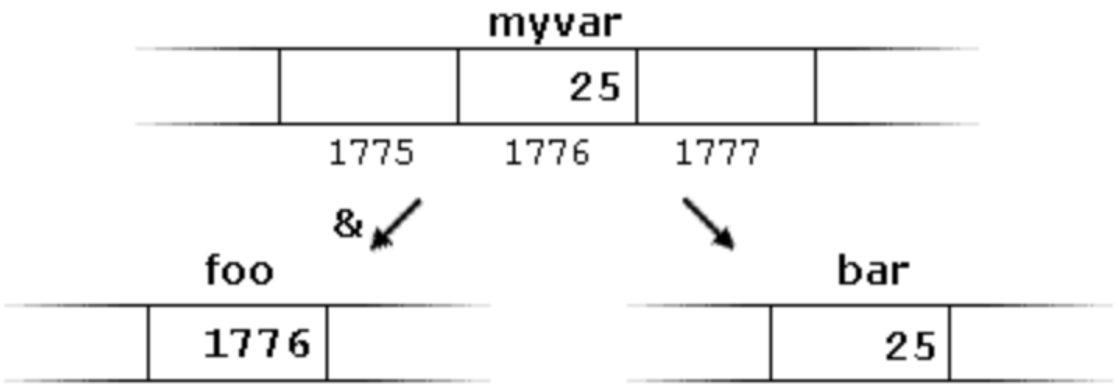
- **use & to get the address of a variable**

```
foo = &myvar;
```

Pointers

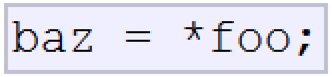


```
myvar = 25;
foo = &myvar;
bar = myvar;
```

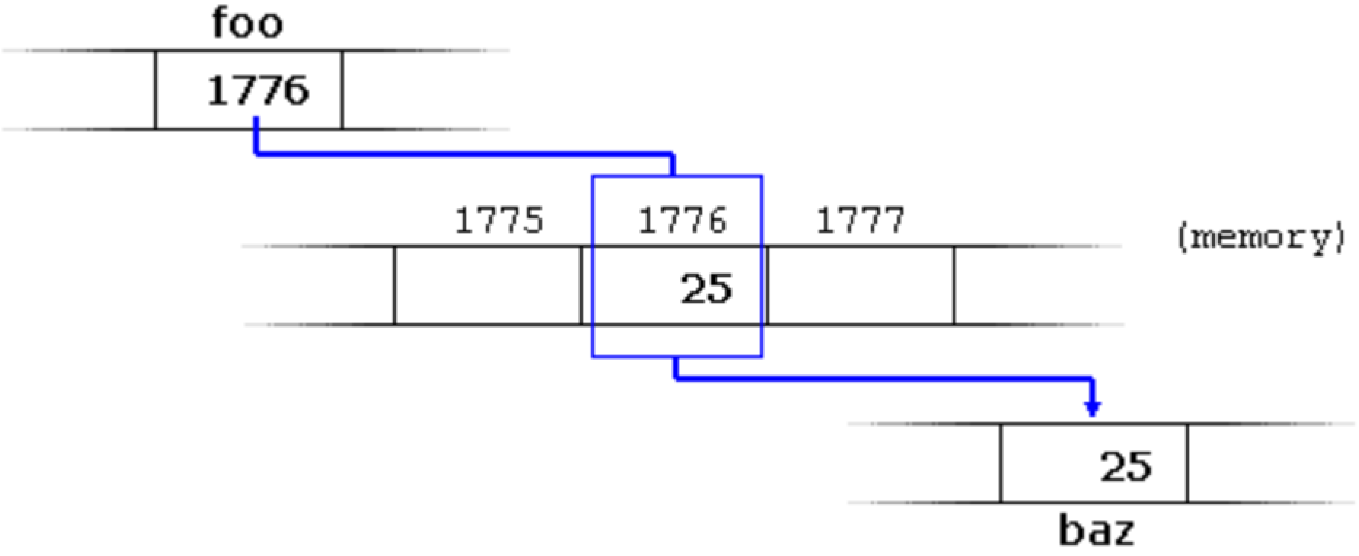


Pointers

- use * to get the value at a pointer (address)



```
baz = *foo;
```



Pointers

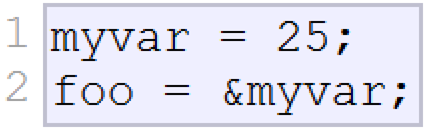
- **& and * are complementary**

- **& means “get the address of”**
  - **p = &c** means the address of c is assigned to the variable p

- *** means “get the value at that address”**
  - termed “dereferencing”
  - **int a = *p** means get the value at the address designated by p and assign it to a
  - ***p = 1** means assign the value of 1 to the memory location designated by the address of p

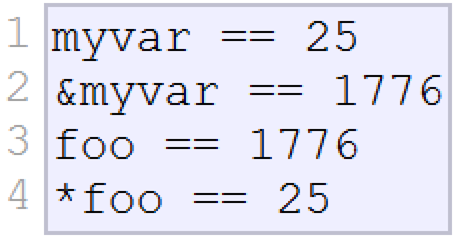
Pointers

- **with following assignments**

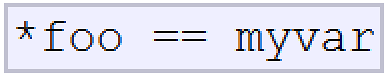


```
myvar = 25;
foo = &myvar;
```

all of the following are true



```
myvar == 25
&myvar == 1776
foo == 1776
*foo == 25
```

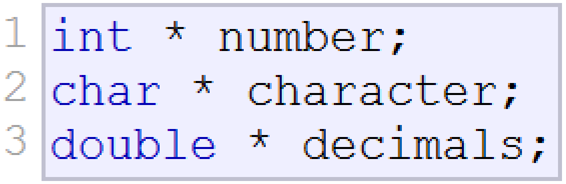


```
*foo == myvar
```

Declaring Pointers

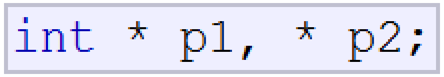
- *** is used in the declaration of a pointer type**
  - **int *p** means variable p is a pointer that points to an integer
  - every pointer points to a specific data type
    - exception: void

- **all pointers are the same size in memory**

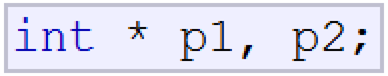


```
int * number;
char * character;
double * decimals;
```

- **different**



```
int * p1, * p2;
```



```
int * p1, p2;
```

Examples

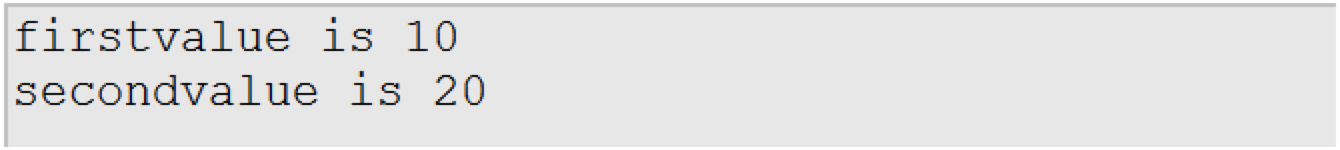
```
#include <stdio.h>

int main()
{
    float i = 10, *j;
    void *k;

    k = &i;
     j = k;

    printf ("%f\n", *j);

    return 0;
}
```

Examples

```
#include <stdio.h>

int main (void)
{
    char ch = 'c';
    char *chptr = &ch;
    int i = 20;
    int *intptr = &i;
    float f = 1.20000;
    float *fptr = &f;
    char *ptr = "I am a string";

    printf ("\n [%c], [%d], [%f], [%c], [%s]\n", *chptr, *intptr, *fptr, *ptr,  ptr);

    return 0;
}
```

Examples

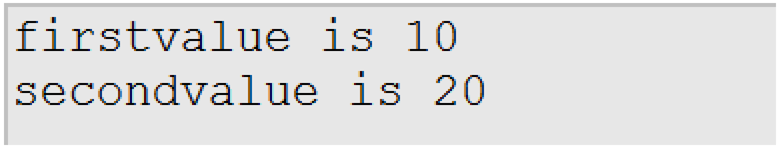
```
#include <stdio.h>

int main ()
{
     int firstvalue, secondvalue;
     int *mypointer;

     mypointer = &firstvalue;
     *mypointer = 10;

     mypointer = &secondvalue;
     *mypointer = 20;

    printf ("firstvalue is %d\n", firstvalue);
    printf (“secondvalue is %d\n", secondvalue);
}
```



```
firstvalue is 10
secondvalue is 20
```

Examples

```
#include <stdio.h>

int main ()
{
    int firstvalue = 5, secondvalue = 15;
    int *p1, *p2;

    p1 = &firstvalue;
    p2 = &secondvalue;

    *p1 = 10;
    *p2 = *p1;
    p1 = p2;
    *p1 = 20;

   printf ("firstvalue is %d\n", firstvalue);
   printf (“secondvalue is %d\n", secondvalue);
}
```



```
firstvalue is 10
secondvalue is 20
```

Pointers

- **if ip points to the integer x (ip = &x) then *ip can occur in any context where x could**
  - ex: *ip = *ip + 10 ⟶ x = x + 10
    - increments the contents of the address at ip by 10

- **unary operators * and & bind more tightly than arithmetic operators**
  - ex: y = *ip + 1 takes whatever ip points at, adds 1, and assigns the result to y
  - other ways to increment by 1:
    - *ip += 1 ⟶ *ip = *ip + 1
    - ++*ip
    - (*ip)++ (the parentheses are necessary because without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left)

Pointers

- **pointers are variables so can be used without dereferencing**
  - ex: int *iq, *ip;

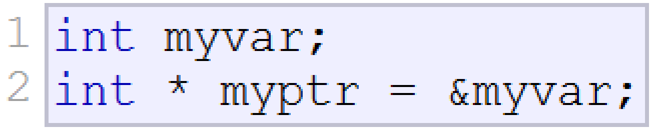
    ```
    iq = ip;
    ```
    - copies the contents of ip (an address) into iq, thus making iq point to whatever ip pointed to

Example

```
#include <stdio.h>

main ()
{
    int x, *p;

    p = &x;
    *p = 0;

    printf ("x is %d\n", x);
    printf ("*p is %d\n", *p);

    *p += 1;

    printf ("x is %d\n", x);

    (*p)++;

    printf ("x is %d\n", x);

    return 0;
}
```

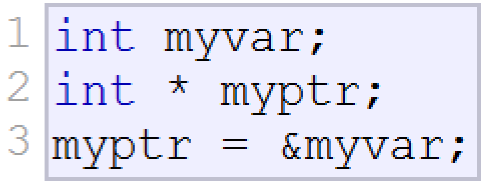
Pointer Initialization

- pointers can be initialized at declaration



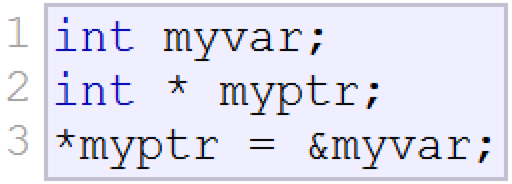
```
int myvar;
int * myptr = &myvar;
```

- same as



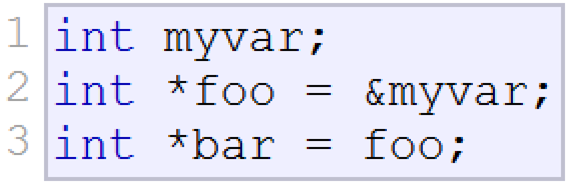
```
int myvar;
int * myptr;
myptr = &myvar;
```

- not valid



```
int myvar;
int * myptr;
*myptr = &myvar;
```

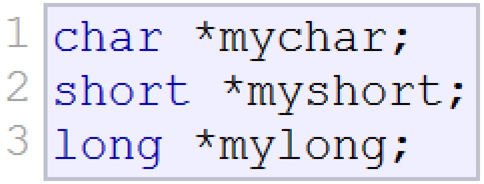
- OK



```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

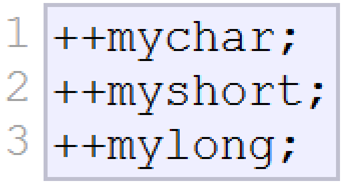
Pointer Arithmetic

- pointers can be used in arithmetic expressions, with underlying size taken into account
- suppose the following have addresses 1000, 2000, 3000



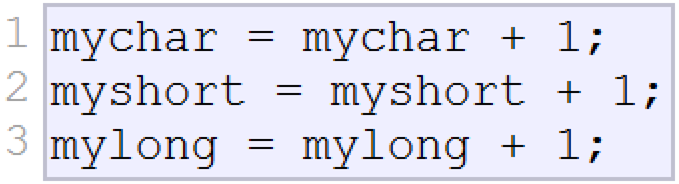
```
char *mychar;
short *myshort;
long *mylong;
```

- after the following



```
++mychar;
++myshort;
++mylong;
```

  - values are 1001, 2002, 3004
- same results for



```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Pointer Arithmetic

- the following is equivalent to *(p++)

```
*p++
```

- other examples

```
*p++    // same as *(p++): increment pointer, and dereference unincremented address
*++p    // same as *(++p): increment pointer, and dereference incremented address
++*p    // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++ // dereference pointer, and post-increment the value it points to
```

- assignment done before increment

```
*p++ = *q++;
```

- same as

```
*p = *q;
++p;
++q;
```

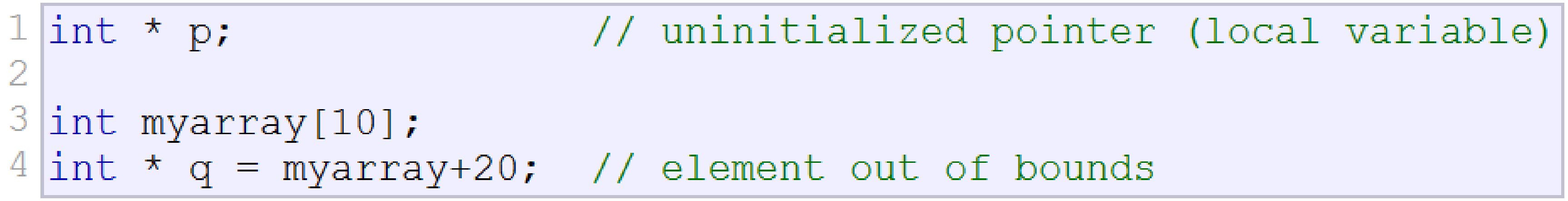
Pointer Arithmetic

- void pointers point to no particular type

```
#include <stdio.h>

void increase (void *data, int psize)
{
   if (psize == sizeof (char))  {
      char &pchar;  pchar = (char *) data;  ++(*pchar);  }
   else if (psize == sizeof (int)) {
      int *pint; pint = (int *(data);  ++(*pint); }
}

int main ()
{
    char a;
    int b = 1602;

    increase (&a, sizeof (a));
    increase (&b, sizeof (b));
    printf (“%d, %d%d\n", a, b);
    return 0;
}
```

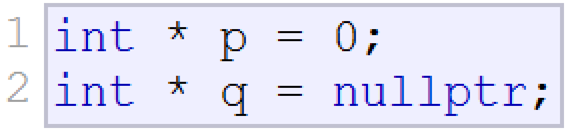
Pointers

- pointers can point to any address



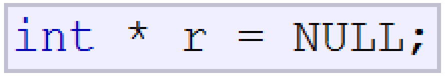
```
int * p;                 // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;   // element out of bounds
```

- pointers can point to nothing



```
int * p = 0;
int * q = nullptr;
```

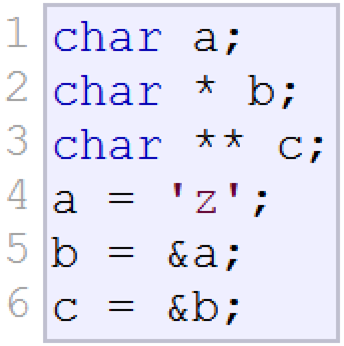
- or simply



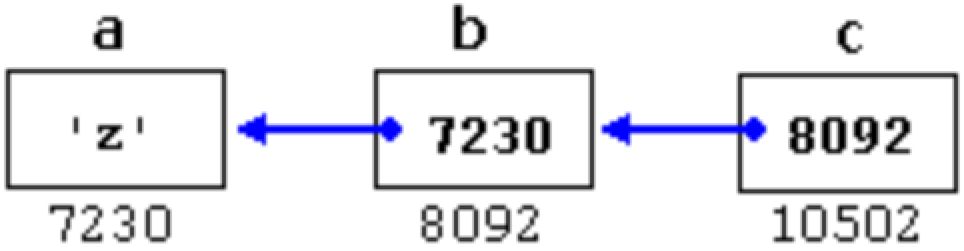
```
int * r = NULL;
```

- NULL pointers and void pointers are different

Pointer to Pointers



```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



Pointers and Arrays

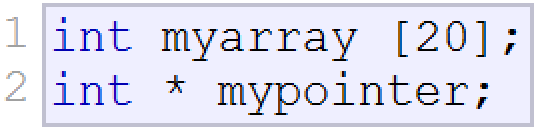
- array name with no index is a pointer to the first element

- the name of the array refers to the whole array; it works by representing a pointer to the start of the array

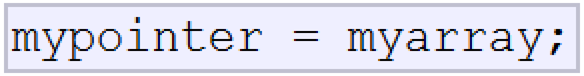
- when passed to functions, an array without any brackets acts like a pointer
  - pass the array directly without using &

Pointers and Arrays

- **arrays can always be converted to pointers**

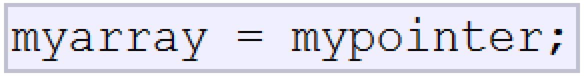


```
int myarray [20];
int * mypointer;
```



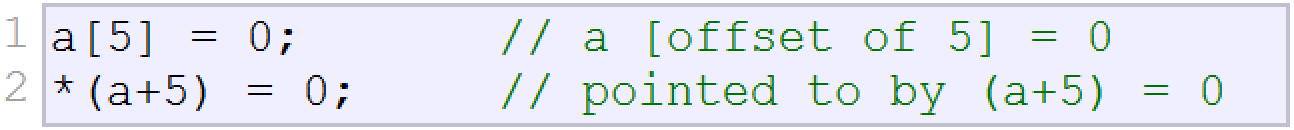
```
mypointer = myarray;
```

- **not valid to go the other way**



```
myarray = mypointer;
```

- **array with index is a simply a pointer with an offset**
  - can be represented with pointer



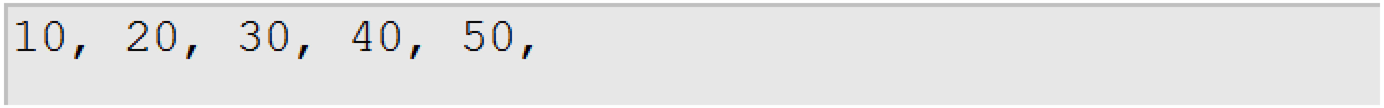
```
a[5] = 0;          // a [offset of 5] = 0
*(a+5) = 0;        // pointed to by (a+5) = 0
```

Pointers and Arrays

```
Prototype/Call

void intSwap (int *x, int *y);
intSwap (&a[i], &a[n – i - 1]);

void printIntArray (int a[], int n);
printIntArray (x, hmny);

int getIntArray (int a[], int nmax, int sentinel);
hmny = getIntArray (x, 10, 0);

void reverseIntArray (int a[], int n);
reverseIntArray (x, hmny);
```

Pointers and Arrays

```
#include <stdio.h>

int main (void)
{
    int numbers [5];
    int *p, n;

    p = numbers;            *p = 10;
    p++;                    *p = 20;
    p = &numbers [2];       *p = 30;
    p = numbers + 3;        *p = 40;
    p = numbers;      *(p + 4) = 50;

    for (n = 0; n < 5; n++)
        printf ("%d, ", numbers [n]);

    return 0;
}
```



```
10, 20, 30, 40, 50,
```

Pointers and Strings

- **a string is an array of characters**
  - no string pointers in C – character pointers instead
  - a pointer to a string holds the address of the first character of the string (just like an array)

- **a string with no index is a memory address without a reference operator (&)**

```
char *ptr;
char str[40];
ptr = str;
```

Pointers and Strings

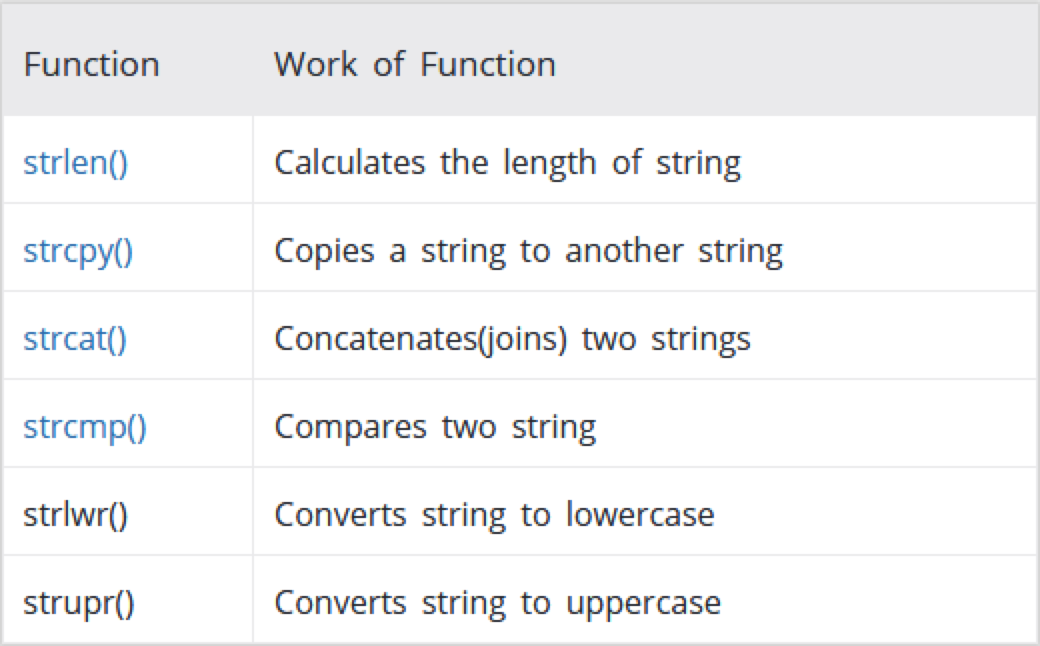
- **strings end with an implied '\0' by default**
  - "I am a string" = I_am_a_string\0
  - sizeof operator returns number of bytes, or characters
  - strlen() function
    - need string.h header file
    - returns the length of the null-terminated string *s* in bytes
      - or, the offset (i.e. starting at position zero) of the terminating null character within the array

```
char string[32] = "hello, world";
sizeof (string) ⇒ 32
strlen (string) ⇒ 12
```

      - this will only work on the character array itself, not a pointer to it

Pointers and Strings

- **summary of string functions**
  - need #include <string.h>



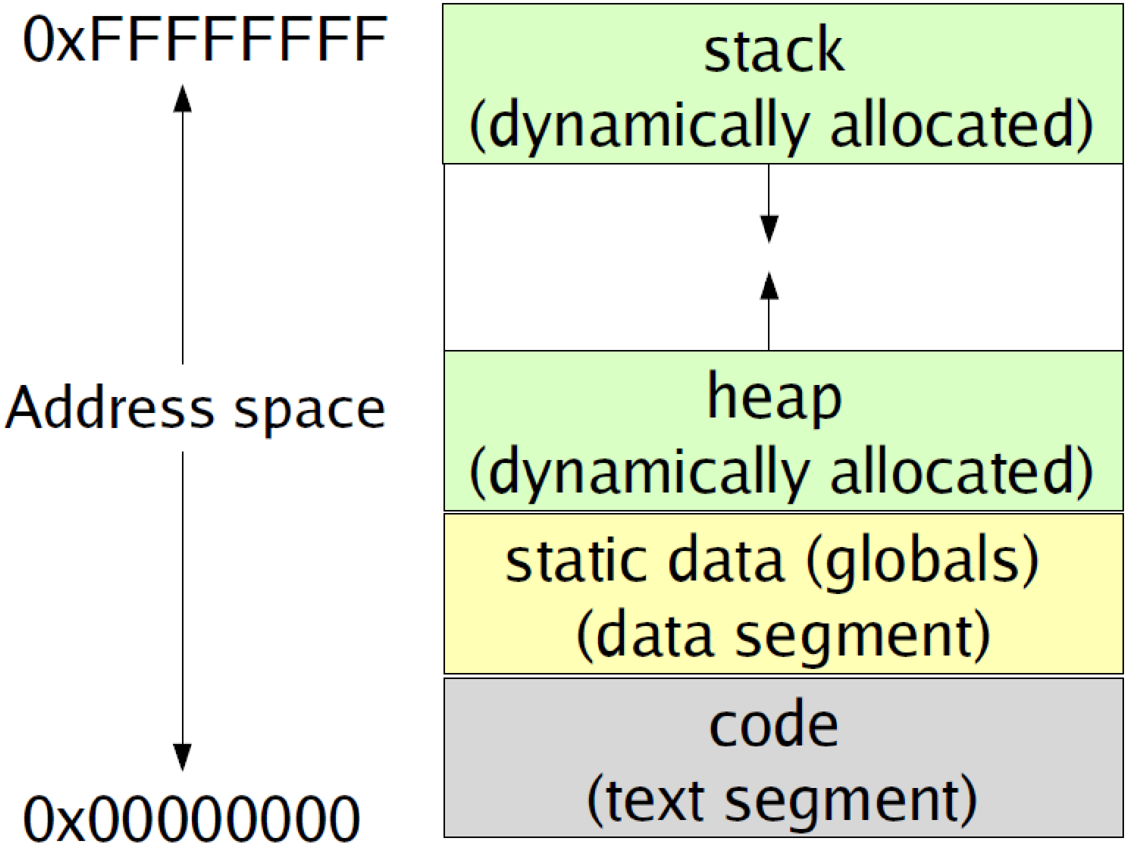
Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

https://www.programiz.com/c-programming/string-handling-functions

Pointers and Strings

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

int main (void)
{
    char arr [4];              // for accommodating 3 characters and one null '\0' byte
    char *ptr = "abc";         // a string containing 'a', 'b', 'c', '\0'

    // reset all the bytes so that none of the bytes contains any junk value
    memset (arr, '\0', sizeof (arr));

    strncpy (arr, ptr, sizeof ("abc"));   // copy the string "abc" into the array arr
    printf ("\n %s \n",arr);              // print the array as string
    arr [0] = 'p';                        // change the first character in the array
    printf ("\n %s \n",arr);              // again print the array as string
    return 0;
}
```

Dynamic Memory Allocation



Address space is just array of 8-bit bytes

Typical total size is: 2^{32}

We will assume that integer is 4 bytes

A *pointer* is just an index into this array

Dynamic Memory Functions

- **found in stdlib.h**

malloc ()	general-purpose memory block
calloc ()	array memory allocation
free ()	de-allocate memory; return to the system

Dynamic Memory Functions: malloc ()

- **malloc () allocates a block of memory**
  - number of bytes passed as argument
  - returns a pointer to that memory if successful
    - NULL otherwise
  - values in memory are uninitialized

- **prototype: void *malloc (size_t size);**
  - size: number of bytes requested
  - returns void* so pointer returned can point to any type of data

Dynamic Memory Functions: malloc ()

- example

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

int main()
{
    int *buffer;

    buffer = (int *) malloc (10 * sizeof (int));
    if (buffer == NULL) {

        printf ("Error allocating memory.\n");
        exit (1);
    }

    free (buffer);
    return 0;
}
```

http://www.codingunit.com/c-reference-stdlib-h-function-malloc

Dynamic Memory Functions: malloc ()

- **another example:**

```
 #include <stdlib.h>

 // set ptr to point to a memory address of size int
int *ptr = (int *) malloc (sizeof (int));

 // slightly cleaner to write malloc statements by taking the size of the
 //   variable pointed to by using the pointer directly
int *ptr = (int *) malloc (sizeof (*ptr));

float *ptr = (float *) malloc (sizeof (*ptr));

float *ptr;
// hundreds of lines of code
ptr = (float *) malloc (sizeof (*ptr));
```

Dynamic Memory Functions: calloc ()

- **calloc () allocates a block of memory**
  - number of items and number bytes per item passed as argument
  - returns a pointer to that memory if successful
    - NULL otherwise
  - values in memory are initialized to zero

- **prototype: void *calloc (size_t num, size_t size);**
  - num: number of items requested
  - size: size of each element
  - returns void* so pointer returned can point to any type of data

Dynamic Memory Functions: calloc ()

- example

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

int main()
{
    int a, n, *buffer;

    scanf ("%d", &a);
    buffer = (int *) calloc (a, sizeof (int));
    if (buffer == NULL) { /* error */ }

    for (n = 0; n < a; n++ ) {
        printf ("Enter number #%d: ", n);
        scanf ("%d", &buffer [n]); }

    printf ("Output: ");
    for (n = 0; n < a; n++)
        printf ("%d", buffer [n]);

    free (buffer);
    return 0;
}
```

malloc () vs. calloc ()

- **number of arguments**
  - malloc () takes a single argument: memory required in bytes
  - calloc () needs two arguments: number of items and size of single item

- **initialization of memory**
  - malloc () does not initialize memory allocated
  - calloc () initializes each element of allocated memory to zero

Dynamic Memory Functions: free ()

- **free () returns allocated memory back to the operating system**
  - pointer to first location in allocated memory passed as argument
  - after freeing a pointer, reset it to NULL

- **prototype: void free (void *p);**
  - p: pointer to memory that will be de-allocated

- **NULL pointer**
  - 0 is assigned to a pointer
  - pointer points to nothing
  - errors can be uncovered immediately when something foolish is done with the pointer (it happens a lot, even with experienced programmers) instead of later, after considerable damage has been done

Structures

- struct

```
typedef struct {
   int weight;
   double price;
} FRUIT_T;

FRUIT_T apple;
FRUIT_T banana, melon;
```

- or

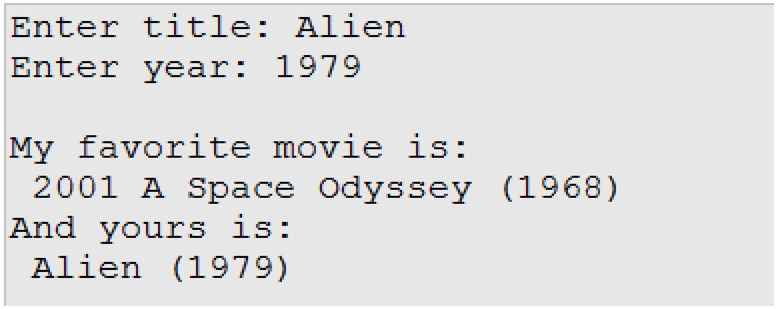
```
struct {
   int weight;
   double price;
} apple, banana, melon;
```

- access

```
apple.weight
apple.price
banana.weight
banana.price
melon.weight
melon.price
```

Structures

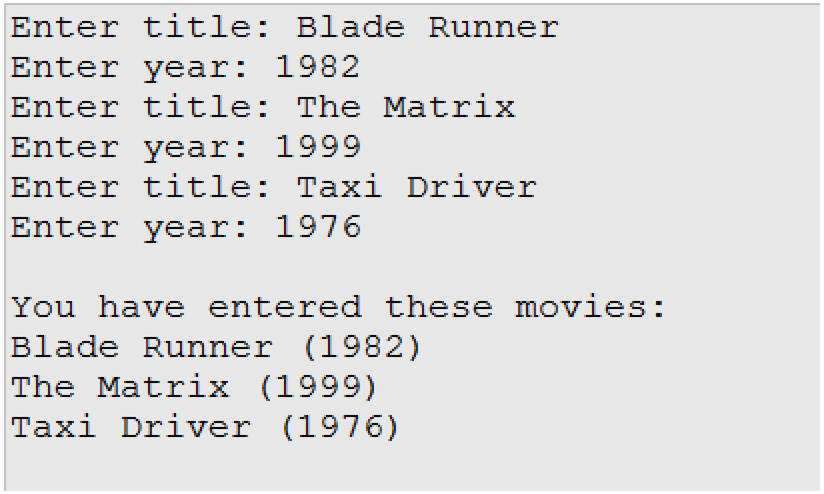
```
#include <stdio.h>
#include <string.h>

typedef struct {
   char title [40];
   int year;
} MOVIE_T;

void print_movie (MOVIE_T movie)
{
   printf ("%s (%d)\n", movie.title, movie.year);
}

int main()
{
   MOVIE_T mine, yours;

   strcpy (mine.title, "2001: A Space Odyssey");
   mine.year = 1968;

   printf ("Enter title: ");
   scanf ("%[^\n]s", yours.title);
   printf ("Enter year: ");
   scanf ("%d", &yours.year);

   printf ("My favorite movie is: ");
   print_movie (mine);
   printf ("And yours is: \n");
   print_movie (yours);
}
```



```
Enter title: Alien
Enter year: 1979

My favorite movie is:
 2001 A Space Odyssey (1968)
And yours is:
 Alien (1979)
```

Structures

```
#include <stdio.h>

typedef struct {
  char title [40];
  int year;
} MOVIE_T;

void print_movie (MOVIE_T movie)
{
  printf ("%s (%d)\n", movie.title, movie.year);
}

int main()
{
  MOVIE_T  films [3];
  int n;

  for (n = 0; n < 3; n++) {
    printf ("Enter title: ");
    scanf (" %[^\n]s", films [n].title);
    printf ("Enter year: ");
    scanf ("%d", &films [n].year);
  }

  printf ("\nYou have entered these movies: \n");
  for (n = 0; n < 3; n++)
    print_movie (films [n]);
}
```



```
Enter title: Blade Runner
Enter year: 1982
Enter title: The Matrix
Enter year: 1999
Enter title: Taxi Driver
Enter year: 1976

You have entered these movies:
Blade Runner (1982)
The Matrix (1999)
Taxi Driver (1976)
```

Pointers to Structures

- pointers to struct

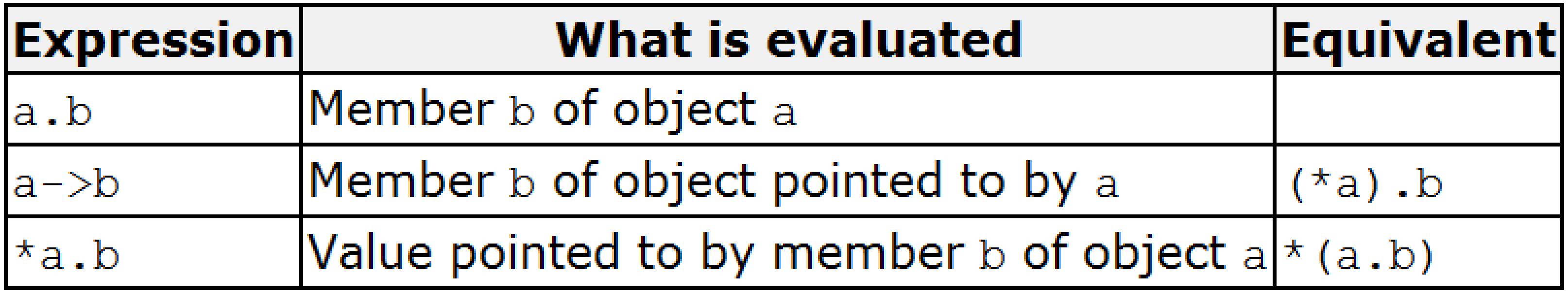
```
pmovie -> title
```

```
(*pmovie).title
```

- different from

```
*pmovie -> title
```

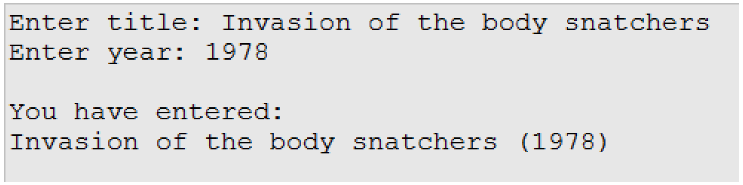
```
*(pmovie -> title)
```



Expression	What is evaluated	Equivalent
`a.b`	Member `b` of object `a`	
`a->b`	Member `b` of object pointed to by `a`	`(*a).b`
`*a.b`	Value pointed to by member `b` of object `a`	`*(a.b)`

Pointers to Structures

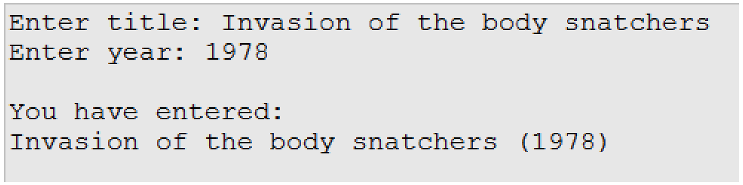
```
#include <stdio.h>

typedef struct {
   char title [40];
   int year;
} MOVIE_T;

int main()
{
   MOVIE_T movie;
   MOVIE_T *pmovie;

   pmovie = &movie;

   printf ("Enter title: ");
   scanf ("%[^\n]s%*c", pmovie -> title);
   printf ("Enter year: ");
   scanf ("%d", &pmovie -> year);

   printf ("\nYou have entered: \n %s (%d)\n",  movie.title, pmovie -> year);
}
```



```
Enter title: Invasion of the body snatchers
Enter year: 1978

You have entered:
Invasion of the body snatchers (1978)
```

Pointers to Structures

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

typedef struct {
   char title [40];
   int year;
} MOVIE_T;

int main()
{
   MOVIE_T *pmovie;

   pmovie = (MOVIE_T *) malloc (sizeof (MOVIE_T));

   printf ("Enter title: ");
   scanf ("%[^\n]s%*c", pmovie -> title);
   printf ("Enter year: ");
   scanf ("%d", &pmovie -> year);

   printf ("\nYou have entered: \n %s (%d)\n",  pmovie -> title, pmovie -> year);
}
```



```
Enter title: Invasion of the body snatchers
Enter year: 1978

You have entered:
Invasion of the body snatchers (1978)
```

Nested Structures

- nested struct

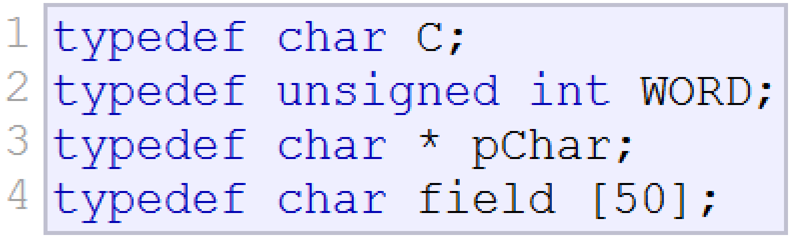
```
typedef struct {
   char title [40];
   int year;
} MOVIE_T;

typedef struct {
   char name [30];
   char email [40];
   MOVIE_T favorite_movie;
} FRIEND_T;

FRIEND_T charlie, maria;
FRIEND_T *pfriend = &charlie;
```

- access

```
charlie.name
maria.favorite_movie
charlie.favorite_movie.year
pfriend -> favorite_movie.year
```

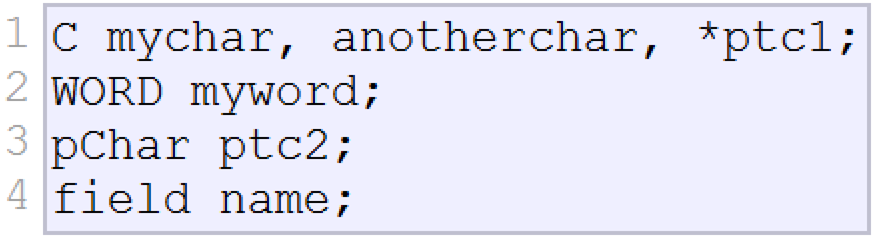
Type Definition

- **define a new type with typedef**



```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

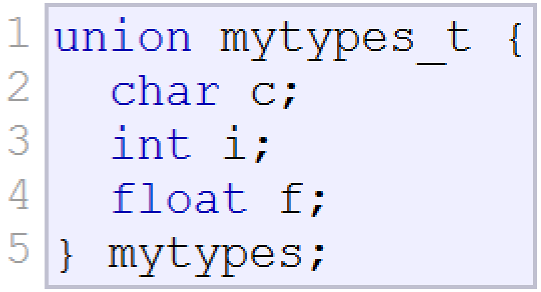
- **define variables with new type**



```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

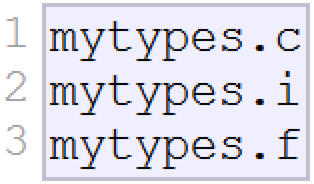
Unions

- **union**
  - similar to a struct, but all fields share the same memory
  - used to save space, or to easily reinterpret bits



```
union mytypes_t {
  char c;
  int i;
  float f;
} mytypes;
```

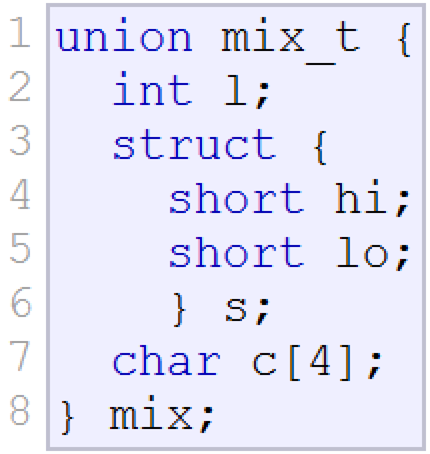
- **access**



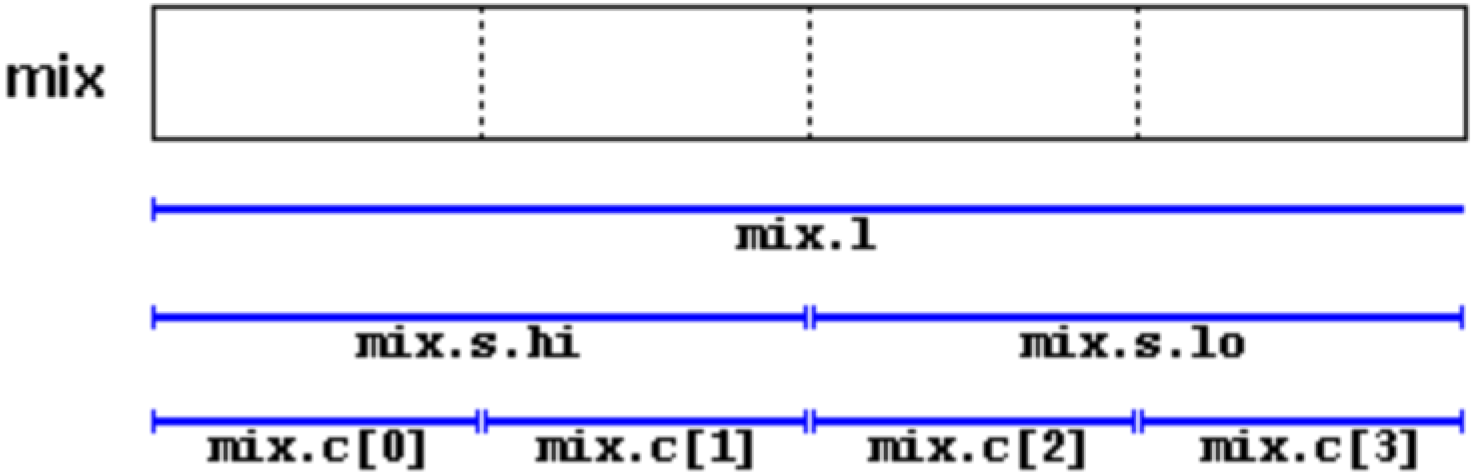
```
mytypes.c
mytypes.i
mytypes.f
```

Unions

- **union**

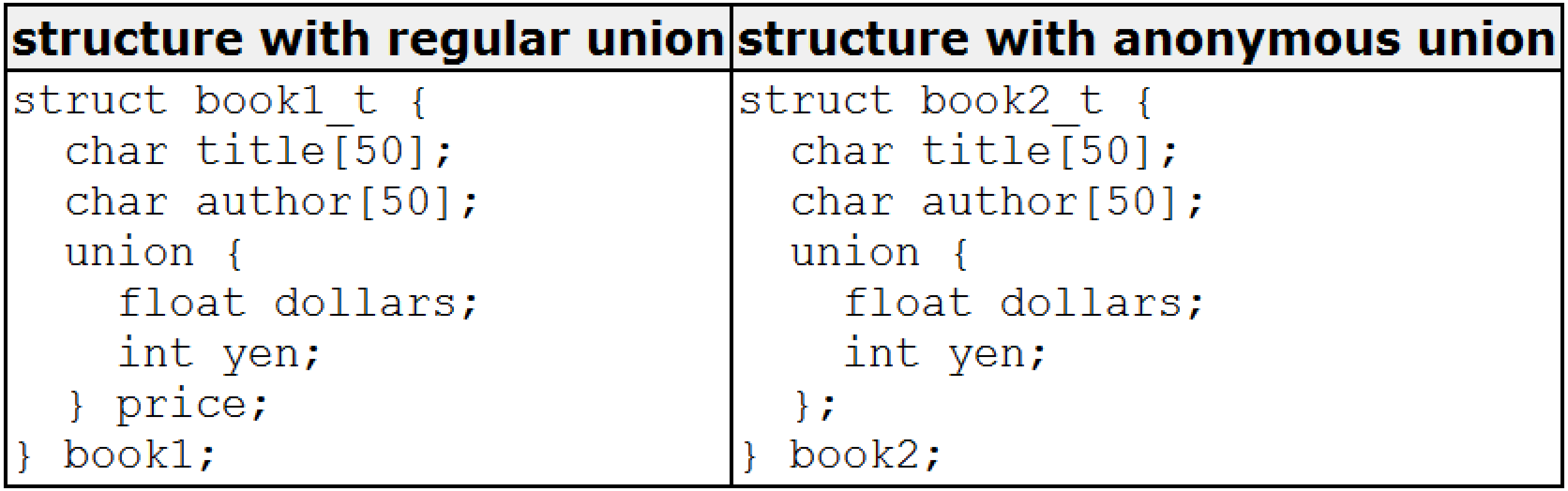


```
union mix_t {
  int l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```



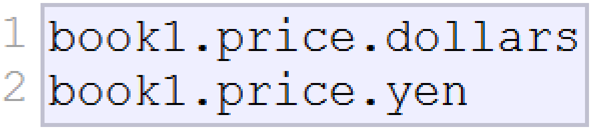
Unions

- anonymous union

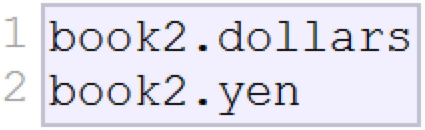


structure with regular union	structure with anonymous union

struct book1_t {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yen;
  } price;
} book1;
	struct book2_t {
	  char title[50];
	  char author[50];
	  union {
	    float dollars;
	    int yen;
	  };
	} book2;



```
book1.price.dollars
book1.price.yen
```



```
book2.dollars
book2.yen
```

Enumerated Types

- **declaration**

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

- **usage**

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

- **alternatively, can assign integer values**
  - by default, starts at 0

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```