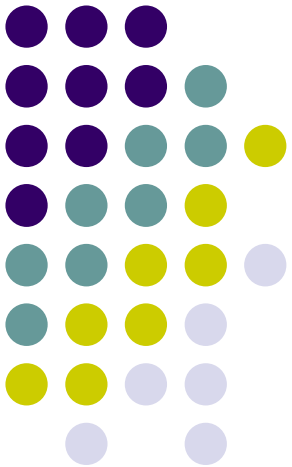
# ECE/ME/EMA/CS 759
# High Performance Computing for Engineering Applications

Quick Overview of C Programming

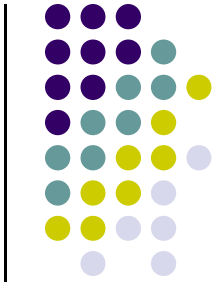
September 6, 2013





"There is no reason for any individual to have a computer in their home."
Ken Olson, president and founder, Digital Equipment Corporation, 1977.

# Before We Get Started…



- Last time
  - Course logistics & syllabus overview
- Today
  - Cover in one lecture what normally covered in two weeks in a regular C course
  - Quick overview of C Programming
    - Essential reading: Chapter 5 of “The C Programming Language” (Kernighan and Ritchie)
    - Read online primer
  - Acknowledgement: Slides on this C Intro include material from D. Zhang & L. Girod
- Other issues:
  - All in the waiting list have been cleared to register
  - You will get forum credentials by Monday end of day if not sooner
  - Learn@UW and class website should be up and running
  - Audio of 2013 will be available for each lecture
  - Video of 2012 to become available during the next week

# The "memory"

Memory: similar to a big table of numbered slots where bytes of data are stored.

The number of a slot is its Address.
One byte Value can be stored in each slot.

Some data values span more than one slot, like the character string "Hello\n"

A Type provides a logical meaning to a span of memory. Some simple types are:

| | |
|---|---|
| `char` | a single character (1 slot) |
| `char [10]` | an array of 10 characters |
| `int` | signed 4 byte integer |
| `float` | 4 byte floating point |
| `int64_t` | signed 8 byte integer |

| Addr | Value |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | |
| 4 | 'H' (72) |
| 5 | 'e' (101) |
| 6 | 'l' (108) |
| 7 | 'l' (108) |
| 8 | 'o' (111) |
| 9 | '\n' (10) |
| 10 | '\0' (0) |
| 11 | |
| 12 | |

# What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Declare a variable by giving it a name and specifying its type and optionally an initial value. There is a subtle difference between “declaring” and “defining” a var.

```
char x;
char y='e';
```

Variable x defined but uninitialized

Initial value

Name

What names are legal?

Type is single character (char)

extern? static? const?

symbol table?

| Symbol | Addr | Value |
|---|---|---|
| | 0 | |
| | 1 | |
| | 2 | |
| | 3 | |
| x | 4 | Some garbage |
| y | 5 | ‘e’ (101) |
| | 6 | |
| | 7 | |
| | 8 | |
| | 9 | |
| | 10 | |
| | 11 | |
| | 12 | |

The compiler puts x and y somewhere in memory.

# Multi-byte Variables

Different types require different amounts of memory. Most architectures store data on "word boundaries", or even multiples of the size of a primitive data type (int, char)

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex

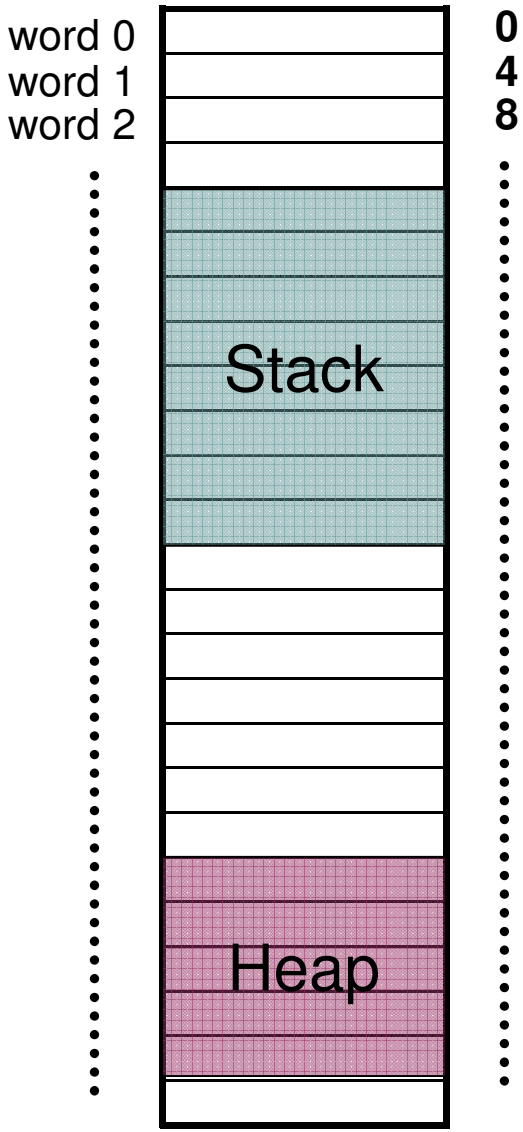
| Symbol | Addr | Value |
|---|---|---|
| | 0 | |
| | 1 | |
| | 2 | |
| | 3 | |
| x | 4 | Some garbage |
| y | 5 | 'e' (101) |
| | 6 | |
| | 7 | |
| z | 8 | 4 |
| | 9 | 3 |
| | 10 | 2 |
| | 11 | 1 |
| | 12 | |

padding

An int requires 4 bytes

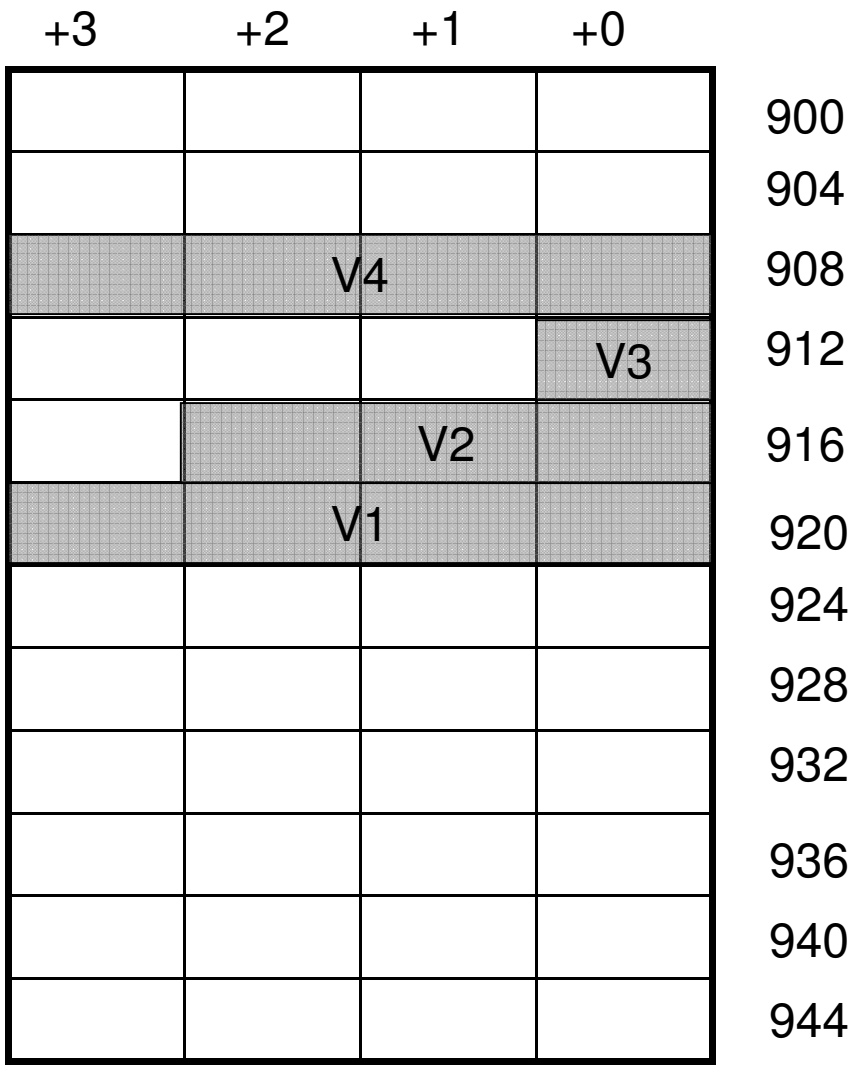
In this picture, the architecture uses little-endian convention, since the least significant byte is stored at the lower address

# Memory, a more detailed view...

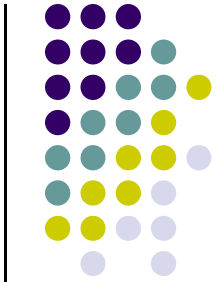



- A sequential list of words, starting from 0.
- Most often, but not always, a word is 4 bytes.
- Local variables are stored on the stack
- Dynamically allocated memory is set aside on the heap (more on this later…)
- For multiple-byte variables, the address is that of the least significant byte (little endian).

# Example...



# Another Example



```
#include <iostream>

int main() {
   char c[10];
   int d[10];
   int* darr;

   darr = (int *)(malloc(10*sizeof(int)));
   size_t sizeC = sizeof(c);
   size_t sizeD = sizeof(d);
   size_t sizeDarr = sizeof(darr);

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

What is the value of:
- sizeC
- sizeD
- sizeDarr

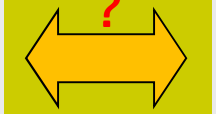
Assume 32 bit OS.

NOTE: ***sizeof*** is a compile-time operator that returns the size, **in multiples of the size of *char***, of the variable or parenthesized type-specifier that it precedes.

# Can a C function modify its arguments?

What if we wanted to implement a function pow_assign() that *modified* its argument? Are these are equivalent?

```
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```



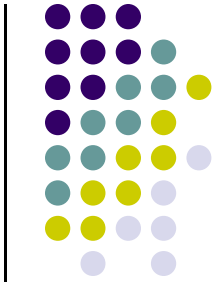

```
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* Is p is 32.0 here ? */
```

Native function, to use you need #include <math.h>

Would this work?

```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; i < exp ; i++) {
    result = result * x;
  }
  x = result;
}
```

# In C you can't change the value of any variable passed as an argument in a function call…



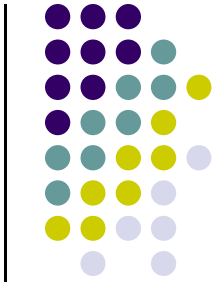
```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; i<exp; i++) {
    result = result * x;
  }
  x = result;
}

// a code snippet that uses above
// function
{
  float p=2.0;
  pow_assign(p, 5);
  // the value of p is 2 here…
}
```

In C, all arguments are passed by value

Keep in mind: pass by value requires the variable to be copied. That copy is then passed to the function. Sometime generating a copy can be expensive…

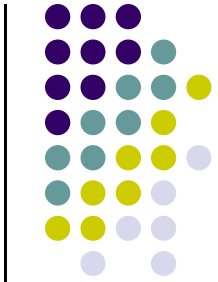
But, what if the argument is the *address* of a variable?

# C Pointers



- What is a pointer?
  - A variable that contains the memory address of another variable or of a function
- In general, it is safe to assume that on 32 bit architectures pointers occupy one word
  - Pointers to int, char, float, void, etc. (“int*”, “char*”, “*float”, “void*”), they all occupy 4 bytes (one word).
- Pointers: *very* many bugs in C programs are traced back to mishandling of pointers…

# Pointers (cont.)



- The need for pointers
  - Modifying a variable (its value) inside a function
    - The pointer to that variable is passed as an argument to the function
  - Passing large objects to functions without the overhead of copying them first
  - Accessing memory allocated on the heap
  - Passing functions as a function argument

# Pointer Validity

A Valid pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior
• Very often the code will crash with a SEGV, that is, Segment Violation, or Segmentation Fault.

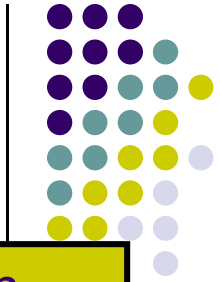
There are two general causes for these errors:
• Coding errors that end up setting the pointer to a strange number
• Use of a pointer that was at one time valid, but later became invalid

Good practice:
•Initialize pointers to 0 (or NULL). NULL is never a valid pointer value, but it is known to be invalid and means "no pointer set".

```
char * get_pointer()
{
  char x=0;
  return &x;
}

{
  char * ptr = get_pointer();
  *ptr = 't';  /* valid? */
}
```

Will *ptr* be valid or invalid?

# Answer: No, it's invalid…



A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is "popped". The pointer will point to an area of the memory that may later get reused and rewritten.

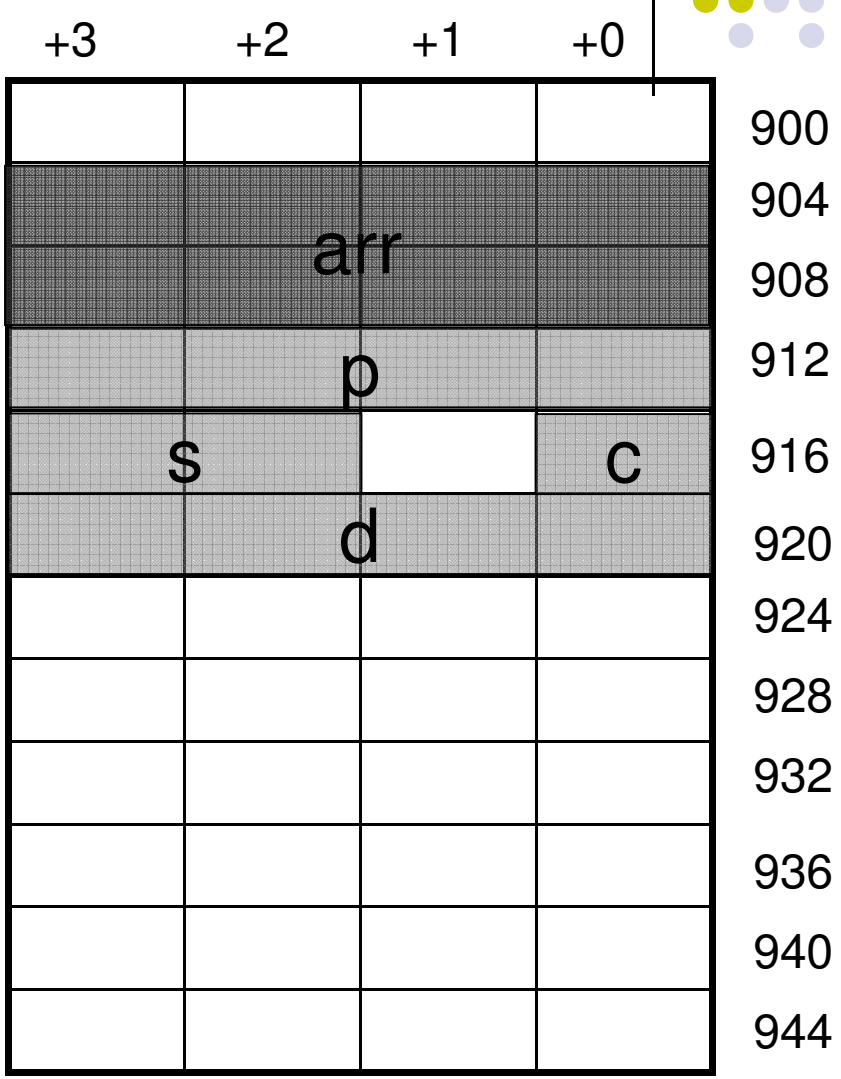
```
char * get_pointer()
{
  char x=0;
  return &x;
}

int main()
{
  char * ptr = get_pointer();
  *ptr = 't';  /* valid? */
  return 0;
}
```

But now, `ptr` points to a location that's no longer in use, and will be reused the next time a function is called!

Here is what I get in VisualStudio when compiling:
main.cpp(3) : warning C4172: returning address of local variable or temporary

# Example: What gets printed out?

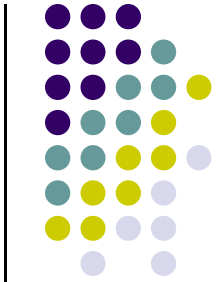
```
int main() {
        int d;
        char c;
        short s;
        int* p;
        int arr[2];
        printf( “%p, %p, %p, %p, %p\n”,&d, &c, &s, &p, arr );
        return 0;
}
```




• NOTE: Here &d = 920 (in practice a 4-byte hex number such as 0x22FC3A08)

**Q: What does get printed out by the *printf* call in the code snippet above?**

# Use of pointers, another example…



- Pass pointer parameters into function

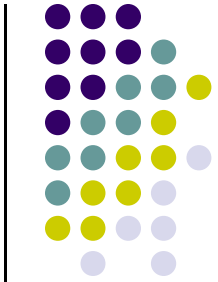
```
void swap(int *px, int *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}
int a = 5;
int b = 6;
swap(&a, &b);
```

- What will happen here?

```
int * a;
int * b;
swap(a, b);
```

```
>> simple.cpp(17) : warning C4700: uninitialized local variable 'b' used
>> simple.cpp(17) : warning C4700: uninitialized local variable 'a' used
```

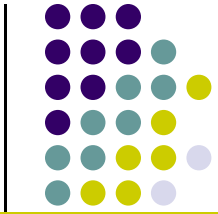
# Dynamic Memory Allocation (Allocation on the Heap)



- Allows the program to determine how much memory it needs *at run time* and to allocate exactly the right amount of storage.
  - It is your responsibility to clean up after you (free the dynamic memory you allocated)

- The region of memory where dynamic allocation and deallocation of memory can take place is called the heap.

# Dynamic Memory Allocation (cont.)



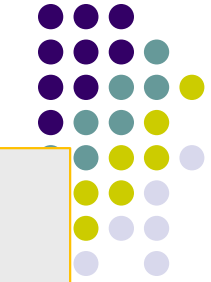
- Functions that come into play in conjunction with dynamic memory allocation

```
void *malloc(size_t number_of_bytes);
        // allocates dynamic memory

size_t sizeof(type);
        // returns the number of bytes of type

void free(void * p)
        // releases dynamic memory allocated
```

- An example of dynamic memory allocation

```
int * ids;    //id arrays
int num_of_ids = 40;
ids = (int*) malloc( sizeof(int) * num_of_ids);
// … do your work here…
free(ids);
```

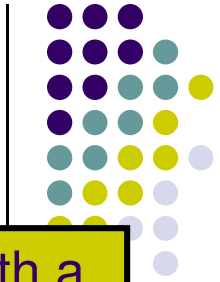
# Exercise



```
#include<iostream>

int main()
{
    double *vals;        //it'll hold some values later on...
    int num_of_vals = 40;
    // vals = (double*) malloc( sizeof(*vals) * num_of_vals);
    vals = (double*) malloc( sizeof(double) * num_of_vals);

    int dummy = sizeof(vals);
    int dummier = sizeof(*vals);

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

- How many bytes were allocated on the heap by the `malloc` operation?
- What is the value of `dummy`?
- What is the value of `dummier`?
- Would you get a compile-time error if you replaced the `malloc` operation by the one that is currently commented out?

# More on Dynamic Memory Allocation



Recall that variables are allocated statically by having declared with a given size. This allocates them in the stack.

Allocating memory at run-time requires dynamic allocation. This allocates them on the heap.

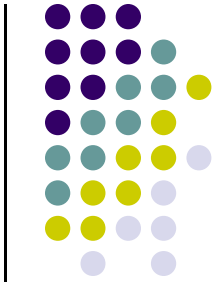
```
int * alloc_ints(size_t requested_count)
{
  int * big_array;
  big_array = (int *)calloc(requested_count, sizeof(int));
  if (big_array == NULL) {
    printf("can't allocate %d ints: %m\n", requested_count);
    return NULL;
  }

  /* big_array[0] through big_array[requested_count-1] are
   * valid and zeroed. */
  return big_array;
}
```

calloc() allocates memory for N elements of size k

Returns NULL if can't alloc

It's OK to return this pointer. It will remain valid until it is freed with free(). However, it's a bad practice to return it (if you need is somewhere else, declare and define it there…)

# Caveats with Dynamic Memory

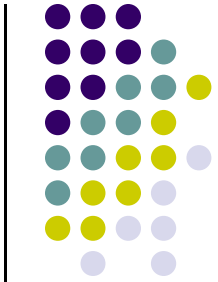


Dynamic memory is useful but be careful when you use it:

- Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()-ed when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory causes a "memory leak".

- Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that was freed. Whenever you free memory you must be certain that you will not try to access it again.

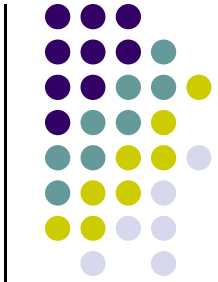
- Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic



Moving on to other topics… What comes next:

- Creating logical layouts of different types (structs)
- Creating new types using typedef
- Using arrays
- Parsing C type names

# Data Structures

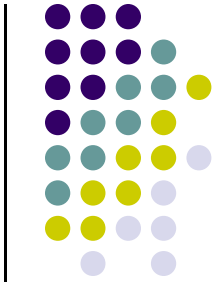


- A data structure is a collection of one or more variables, possibly of different types.

- An example of student record

```
struct StudRecord {
        char name[50];
        int id;
        int age;
        int major;
};
```

# Data Structures (cont.)



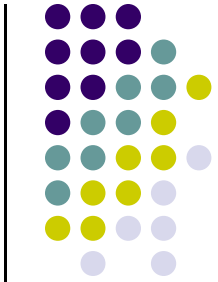
- A data structure is also a data type

```
struct StudRecord my_record;
struct StudRecord * myPointer;
myPointer = & my_record;
```

- Accessing a field inside a data structure

```
my_record.id = 10;
        \\ or
myPointer->id = 10;
```

# Data Structures (cont.)



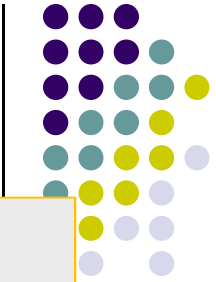
- Allocating a data structure instance

This is a new type now

```
struct StudRecord* pStudentRecord;
pStudentRecord = (StudRecord*)malloc(sizeof(struct StudRecord));
pStudentRecord ->id = 10;
```

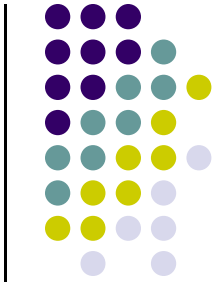
- IMPORTANT: Never calculate the size of data structure yourself. Rely on the sizeof() function.

# Example



```
#include<iostream>

int main() {
    struct StudRecord {
        char name[50];
        int id;
        int age;
        int major;
    };

    StudRecord* pStudentRecord;
    pStudentRecord = (StudRecord*)malloc(sizeof(StudRecord));

    strcpy(pStudentRecord->name, "Joe Doe");
    pStudentRecord->id = 903107;
    pStudentRecord->age = 20;
    pStudentRecord->major = 643;

    return 0;
}
```

# Example:
# Use of malloc, memmove, struct



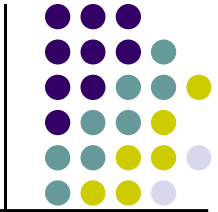
```
#include<iostream>

int main(){
    struct my_struct {
        int counter;
        float average;
        int in_use;
    } init;

    init.counter = 2;
    init.in_use = 0;
    init.average = 1.0f;

    //allocate heap memory for another my_struct variable
    my_struct* s;
    s = (my_struct*)malloc(sizeof(my_struct));
    memmove(s, &init, sizeof(init));

    return 0;
}
```

# The “typedef” concept

Using typedef to improve readability…

```
struct StudRecord {
  char name[50];
  int id;
  int age;
  int major;
};

typedef struct StudRecord RECORD;

int main()
{
   RECORD   my_record;
   strcpy_s(my_record.name, “Jean Doe");
   my_record.age = 21;
   my_record.id = 6114;

   RECORD* p = &my_record;
   p->major = 643;
   return 0;
}
```

# Arrays



Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

```
/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* access element 0, change its value */
x[0] = 'T';

/* pointer arithmetic to get 4th entry */
char elt3 = *(x+3);  /* x[3] */

/* x[0] evaluates to the first element;
 * x evaluates to the address of the
 * first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
  /* process y[i] */
  printf("%c\n", y[i]);
}
```

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0..9)

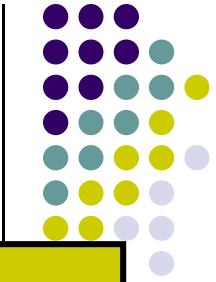
x[3] == *(x+3) == 't'  (*notice, it's not* 's'!)

For loop that iterates from 0 to COUNT-1.

| Symbol | Addr | Value |
|---|---|---|
| char x [0] | 100 | 't' |
| char x [1] | 101 | 'e' |
| char x [2] | 102 | 's' |
| char x [3] | 103 | 't' |
| char x [4] | 104 | '\0' |

**Q**: What's the difference between "char x[5]" and a declaration like "char *x"?

# How to Parse and Define C Types

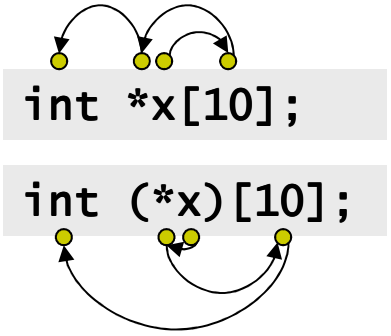


At this point we have seen a few basic types, arrays, pointer types, and structures. So far we've glossed over how types are named.

```
int x;         /* int;                          */  typedef int T;
int *x;        /* pointer to int;               */  typedef int *T;
int x[10];     /* array of ints;                */  typedef int T[10];
int *x[10];    /* array of pointers to int;     */  typedef int *T[10];
int (*x)[10]; /* pointer to array of ints;      */  typedef int (*T)[10];
```

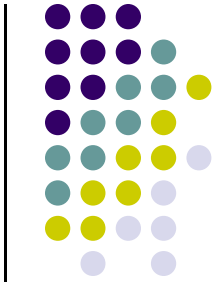
Here's how you typedef this…

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:




Arrays are the primary source of confusion. When in doubt, use extra parens to clarify the expression.

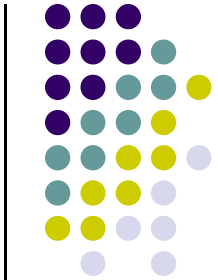
# Example



```
#include<iostream>

int main()
{
    int x[3]={9, -3, 2};
    int (*y)[3];

    y = &x;

    (*y)[0] = 7;
    (*y)[2] = 5;

    return 0;
}
```

- What are the values in x right before the return statement?

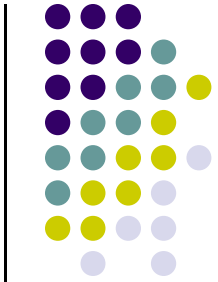
# Rules of Precedence



- How do evaluate x + 3*y[2]?
  - According to the rules of precedence listed below, from highest

| Category | Operators |
|---|---|
| Primary | `x.y  f(x)  a[x]  x++  x--  new typeof checked  unchecked` |
| Unary | `+  -  !  ~  ++x  --x  (T)x` |
| Multiplicative | `*  /  %` |
| Additive | `+  -` |
| Shift | `<<  >>` |
| Relational and type testing | `<  >  <=  >=  is  as` |
| Equality | `==  !=` |
| Logical | `AND &` |
| Logical | `XOR ^` |
| Logical | `OR \|` |
| Conditional | `AND &&` |
| Conditional | `OR \|\|` |
| Conditional | `?:` |
| Assignment | `=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  \|=` |

# Function Types



Another less obvious construct is the "**pointer to function**" type.
For example, qsort: (a sort function in the standard library)

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *) );
```

The last argument is a comparison function

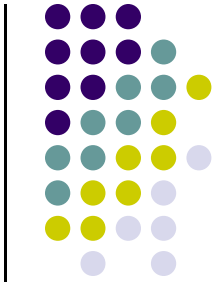
```
/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type) (const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

const means the function is not allowed to modify memory via this pointer.

size_t is an unsigned int

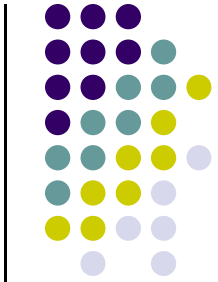
void * is a pointer to memory of unknown type.

# Why is Software Development Hard?

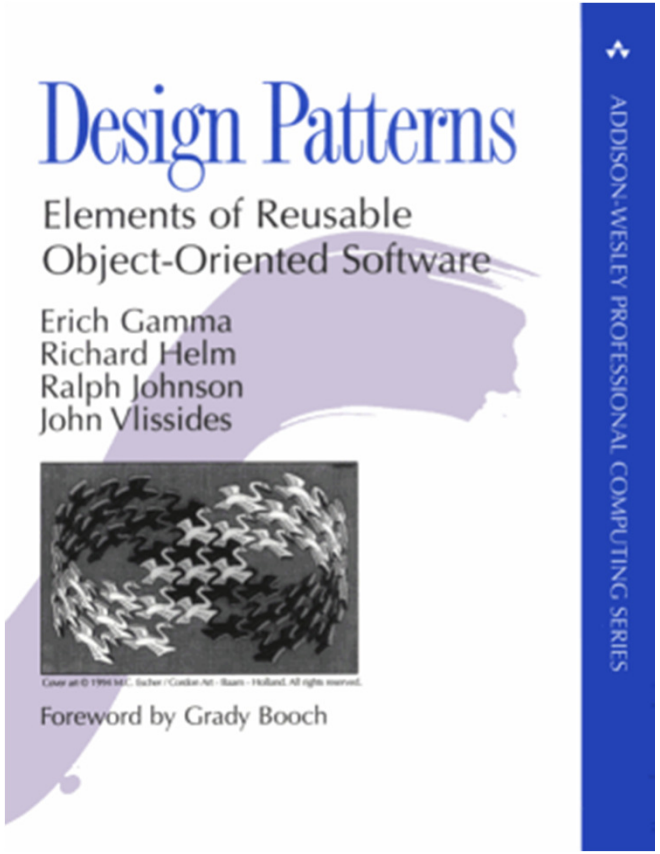


- Complexity: Every conditional ("if") doubles the number of paths through your code, every bit of state doubles possible states
  - Recommendation: reuse code with functions, avoid duplicate state variables
- Mutability: Software is easy to change.. Great for rapid fixes… And rapid breakage…
  - Recommendation: tidy, readable code, easy to understand by inspection, provide *plenty* of meaningful comments.
- Flexibility: Problems can be solved in many different ways. Few hard constraints, easy to let your horses run wild
  - Recommendation: discipline and use of design patterns

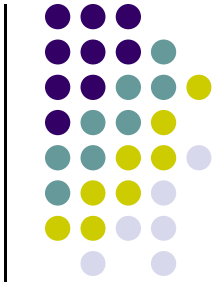
# Design Patterns



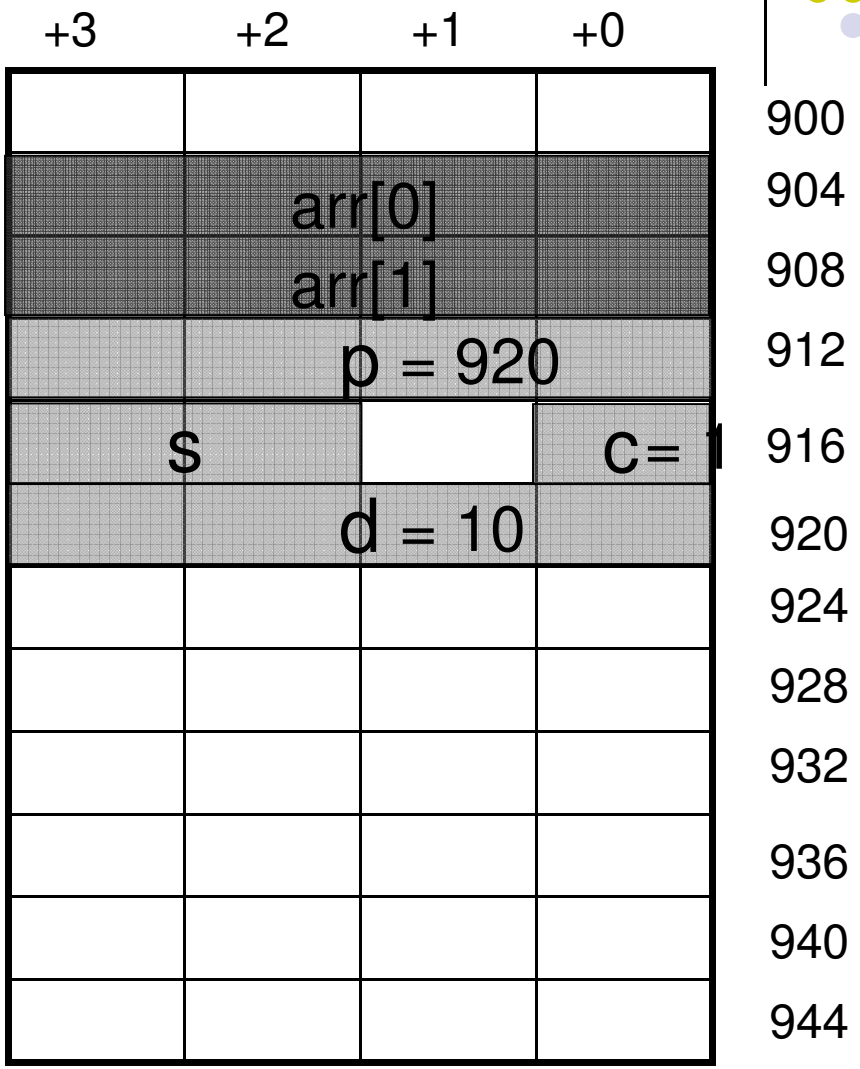
- A really good book if you are serious about this…



# Quiz:
# Usage of Pointers & Pointer Arithmetic



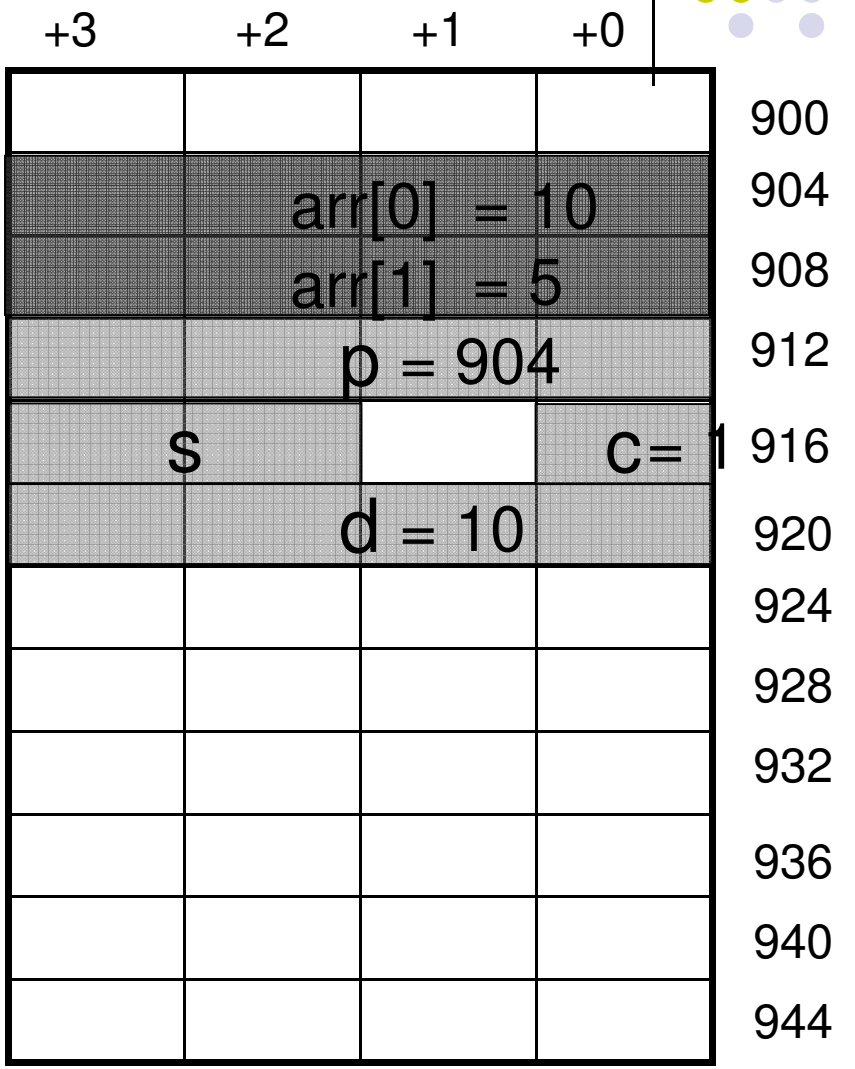
```
int main() {
      int d;
      char c;
      short s;
      int* p;
      int arr[2];

      p = &d;
      *p = 10;
      c = (char)1;

      p = arr;
      *(p+1) = 5;
      p[0] = d;

      *( (char*)p + 1 ) = c;

      return 0;
}
```




**Q: What are the values stored in arr? [assume little endian architecture]**

# Quiz [Cntd.]

```
p = &d;
*p = 10;
c = (char)1;

p = arr;
*(p+1) = 5;  // int* p;
p[0] = d;

*( (char*)p + 1 ) = c;
```

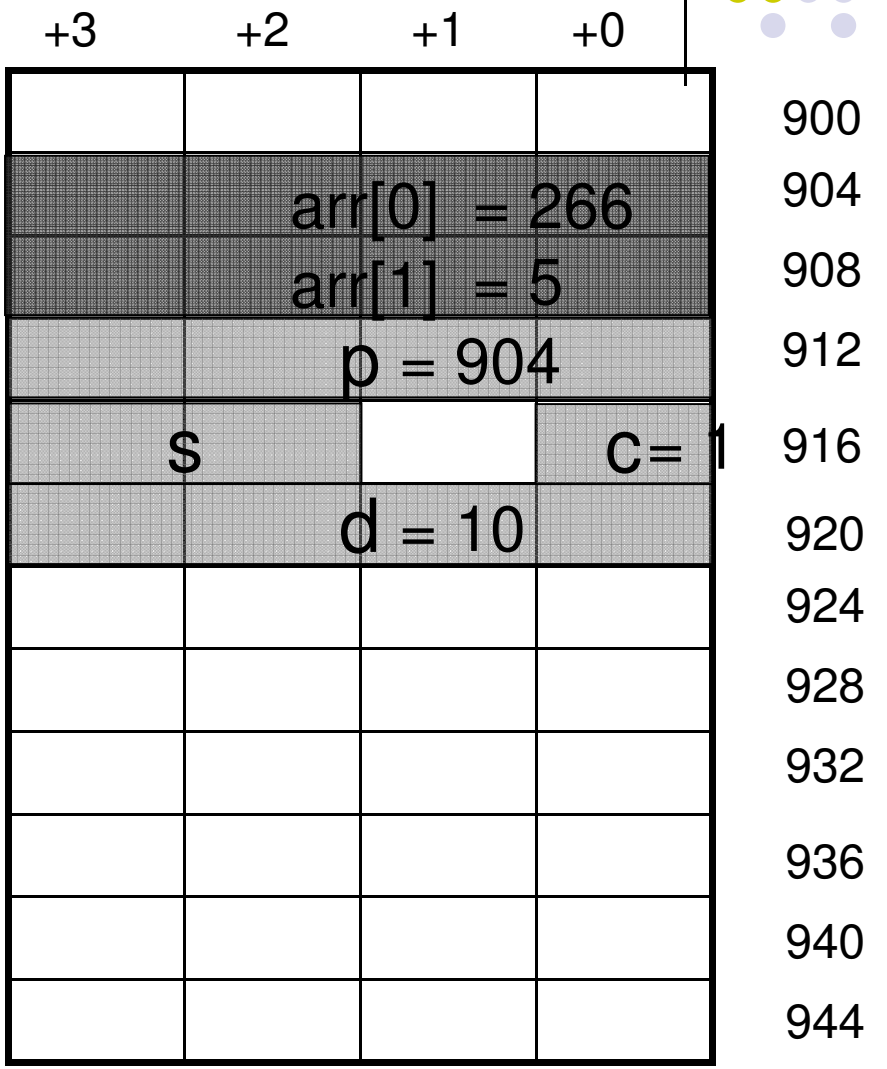



***Question: arr[0] = ?***

# Quiz [Cntd.]

```
p = (int*) malloc(sizeof(int)*3);
p[2] = arr[1] * 3;
s = (short)( *(p+2) );
free( p );
p=NULL;
```

Assumption: we have the same memory layout like in the previous example



# Quiz [Cntd.]

```
p = (int*) malloc(sizeof(int)*3);
p[2] = arr[1] * 3;
short s = (short)( *(p+2) );
free( p );
```

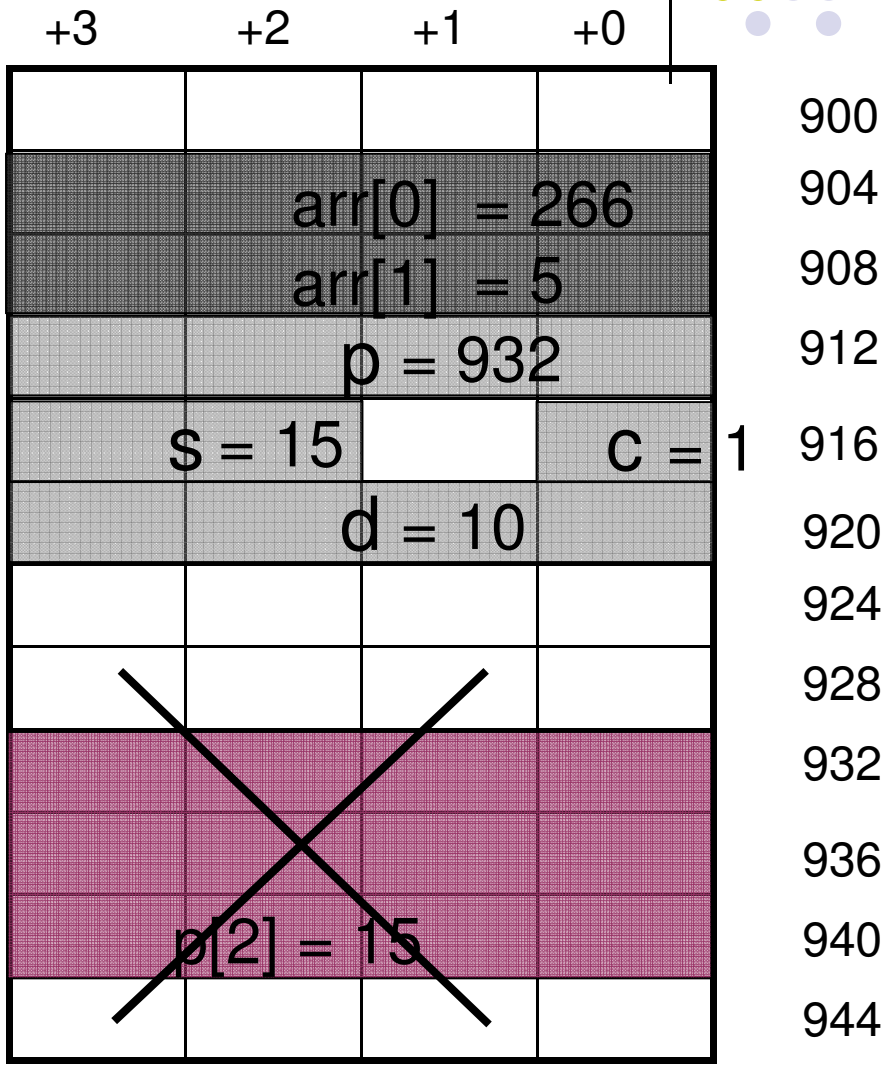
Q: what will be the value of “s”, and why?

Q: what if you say p[2]=0 after you free the memory?

A: run time error.

Q: what if you do not call free(p)?

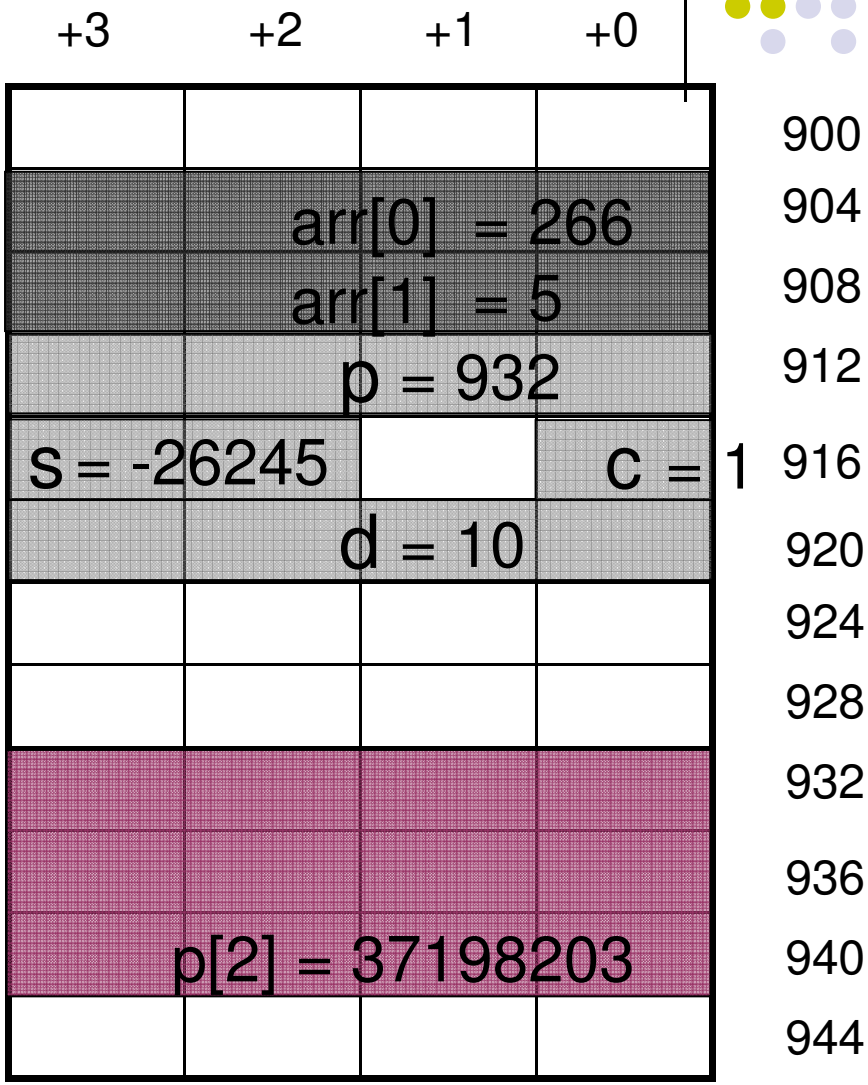
A: memory leak.



# Quiz [Cntd.]

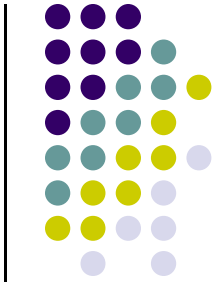
```
int dummy = 12399401
p = (int*) malloc(sizeof(int)*3);
p[2] = dummy * 3;
s = (short)( *(p+2) );
free( p );
```

Q: what is the value of "s" now, and why?




A:

| | | |
|---|---|---|
| p,3 | 932 | int * |
| [0] | -842150451 | int |
| [1] | -842150451 | int |
| [2] | 37198203 | int |
| s | -26245 | short |



## End: Overview of C

## Beginning: Discussion of Hardware Trends