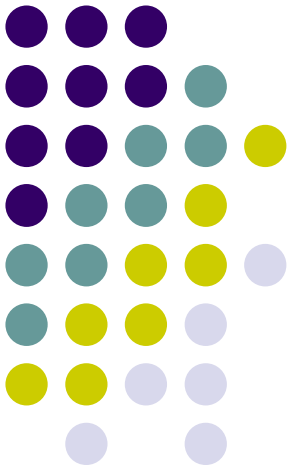
ME964
High Performance Computing for Engineering Applications

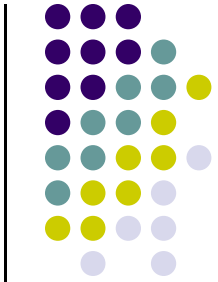
Quick Overview of C Programming

January 20, 2011



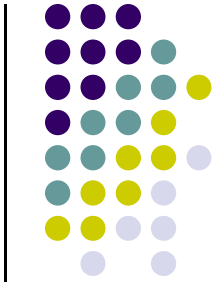
"There is no reason for any individual to have a computer in their home."
Ken Olson, president and founder, Digital Equipment Corporation, 1977.

Before We Get Started…



- Last time
  - Course logistics & syllabus overview
  - Discussed Midterm Projects
    - Discrete Element Method on the GPU
    - Collision Detection on the GPU
    - Basic Finite Element Analysis on the GPU
    - Sparse Linear Solver on the GPU
- Today
  - Quick overview of C Programming
    - Essential read: Chapter 5 of “The C Programming Language” (Kernighan and Ritchie)
  - Acknowledgement: Slides on this C Intro include material due to Donghui Zhang and Lewis Girod
- Correction:
  - Email your homework a this address: me964uw@gmail.com

Auditing the Course



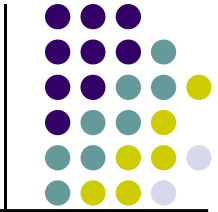
- Why auditing?
  - Large participation justifies another offering of this course
  - Augments your experience with this class
    - You can get an account on the GPU cluster
    - You will be added to the email list
    - Can post questions on the forum

- How to register for auditing:
  - In order to audit a course, a student must first enroll in the course as usual. Then the student must request to audit the course online. (There is a tutorial available through the Office of the Registrar.) Finally, the student must save & print the form. Once they have obtained the necessary signatures, the form should be turned in to the Academic Dean in the Grad School at 217 Bascom. The Grad School offers more information on Auditing Courses in their Academic Policies and Procedures.

Tutorial website: http://www.registrar.wisc.edu/isis_helpdocs/enrollment_demos/V90CourseChangeRequest/V90CourseChangeRequest.htm

Auditing Courses: http://www.grad.wisc.edu/education/acadpolicy/guidelines.html#13

C Syntax and Hello World



#include inserts another file. ".h" files are called "header" files. They contain declarations/definitions needed to interface to libraries and code in other ".c" files.

What do the < > mean?

A comment, ignored by the compiler

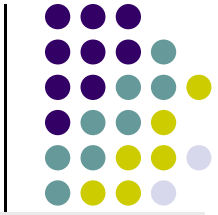
```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

The main() function is always where your program starts running.

Blocks of code ("lexical scopes") are marked by { … }

Return '0' from this function

Lexical Scoping



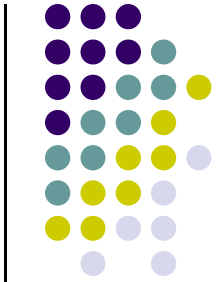
Every Variable is Defined within some scope. A Variable cannot be referenced by name (a.k.a. Symbol) from outside of that scope.

Lexical scopes are defined with curly braces { }.

- The scope of Function Arguments is the complete body of that function.
- The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block
- The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called "Global" Vars.

```
void p(char x)
{
                /* p,x */
  char y;
                /* p,x,y */
  char z;
                /* p,x,y,z */
}
                /* p */
char z;
                /* p,z */

void q(char a)
{
  char b;
                /* p,z,q,a,b */

  {                                  char b?
    char c;
                /* p,z,q,a,b,c */
  }
                                     legal?
  char d;
  /* p,z,q,a,b,d (not c) */
}

/* p,z,q */
```

Comparison and Mathematical Operators



```
== equal to
<  less than
<= less than or equal
>  greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
!  logical not
```

```
+  plus
-  minus
*  mult
/  divide
%  modulo
```

```
&  bitwise and
|  bitwise or
^  bitwise xor
~  bitwise not
<< shift left
>> shift right
```

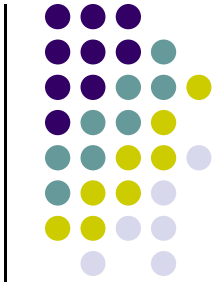
Beware division:

- 5 / 10 → 0 *whereas* 5 / 10.0 → 0.5
- Division by 0 will cause a FPE

Don't confuse & and &&..
1 & 2 → 0 *whereas* 1 && 2 → <true>

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit.

Assignment Operators



```
x = y    assign y to x
x++      post-increment x
++x      pre-increment x
x--      post-decrement x
--x      pre-decrement x
```

```
x += y   assign (x+y) to x
x -= y   assign (x-y) to x
x *= y   assign (x*y) to x
x /= y   assign (x/y) to x
x %= y   assign (x%y) to x
```

Note the difference between ++x and x++ (high vs low priority (precedence)):

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

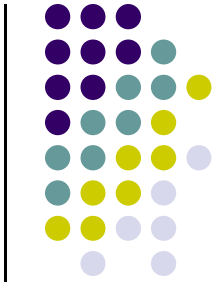
```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don’t confuse “=“ and “==“!

```
int x=5;
if (x==6)   /* false */
{
  /* ... */
}
/* x is still 5 */
```

```
int x=5;
if (x=6)   /* always true */
{
  /* x is now 6 */
}
/* ... */
```

A Quick Digression About the Compiler



```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Preprocess

```
__extension__ typedef  unsigned long long int
__dev_t;
__extension__ typedef  unsigned int   __uid_t;
__extension__ typedef  unsigned int   __gid_t;
__extension__ typedef  unsigned long int
__ino_t;
__extension__ typedef  unsigned long long int
__ino64_t;
__extension__ typedef  unsigned int
__nlink_t;
__extension__ typedef  long int   __off_t;
__extension__ typedef  long long int
__off64_t;
extern void flockfile (FILE *__stream)  ;
extern int ftrylockfile (FILE *__stream)  ;
extern void funlockfile (FILE *__stream)  ;
int main(int argc, char **argv)
{
  printf("Hello World\n");
  return 0;
}
```

Compile

my_program

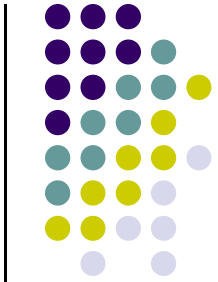
Compilation occurs in two steps: "Preprocessing" and "Compiling"

In Preprocessing, source code is "expanded" into a larger form that is simpler for the compiler to understand. Any line that starts with '#' is a line that is interpreted by the Preprocessor.

- Include files are "pasted in" (#include)
- Macros are "expanded" (#define)
- Comments are stripped out (/* */ , //)
- Continued lines are joined (\ )

The compiler then converts the resulting text (called translation unit) into binary code the CPU can execute.

C Memory Pointers



- To discuss memory pointers, we need to talk a bit about the concept of memory

- We'll conclude by touching on a couple of other C elements:
  - Arrays, typedef, and structs

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

declare vs. define

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

Variable x declared but undefined

Initial value

Name

What names are legal?

Type is single character (char)

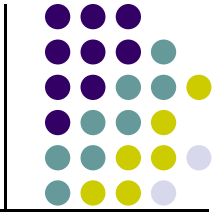
extern? static? const?

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

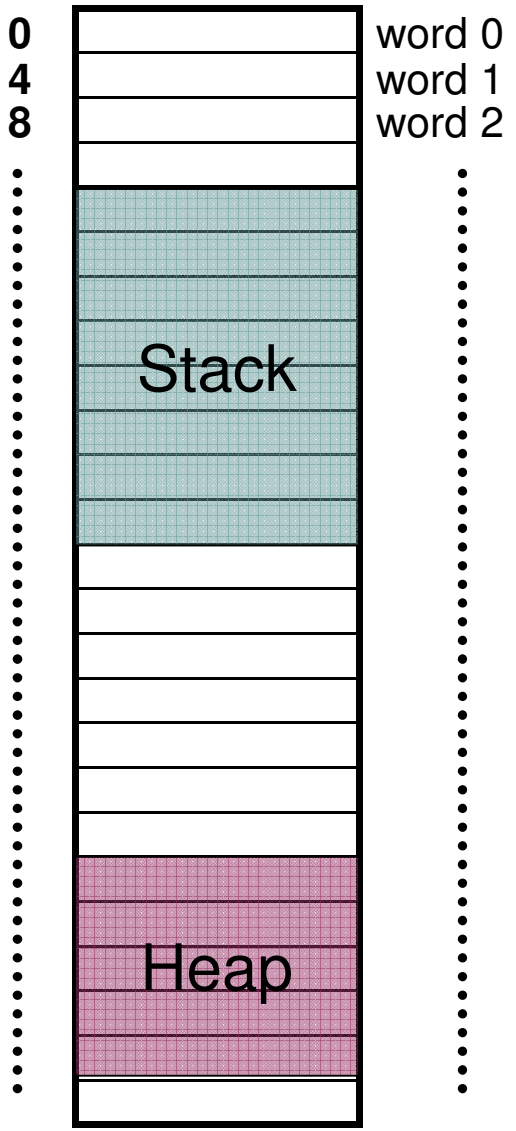
padding

An int requires 4 bytes

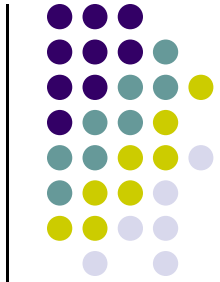
Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	Some garbage
y	5	'e' (101)
	6	
	7	
z	8	4
	9	3
	10	2
	11	1
	12	

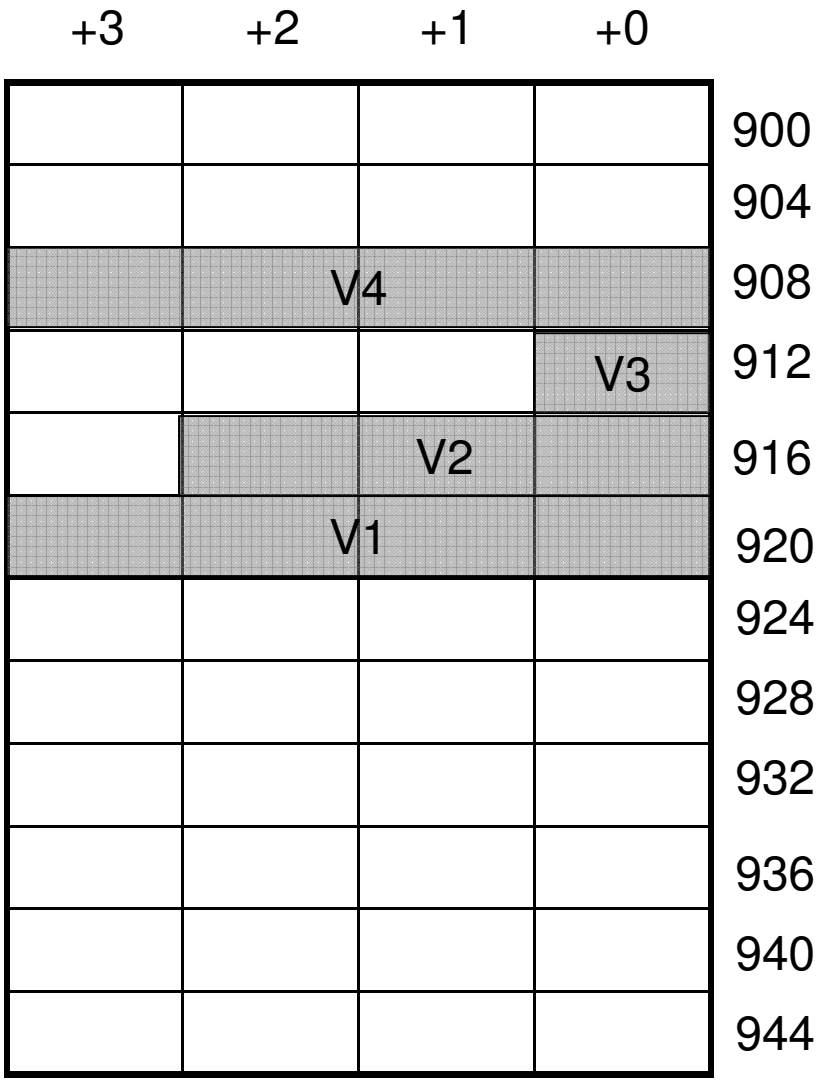
Memory, a more detailed view...

- A sequential list of words, starting from 0.
- On 32bit architectures (e.g. Win32): each word is 4 bytes.
- Local variables are stored in 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 smallest byte (little endian).

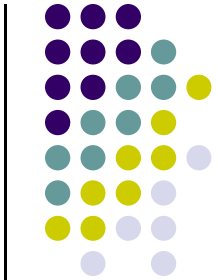


Example...





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;
}
```

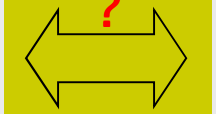
What is the value of:
- sizeC
- sizeD
- sizeDarr

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, so that these are equivalent:

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



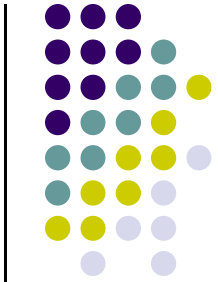

```
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* Is p is 32.0 here ? */
```

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…



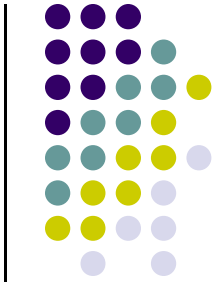
```
void pow_assign(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  x = result;
}

// 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…

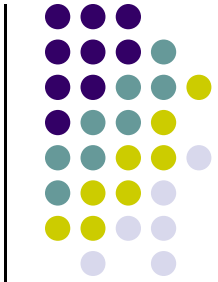
But, what if the argument is the *address* of a variable?

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
  - Needed when you want to modify a variable (its value) inside a function
    - The pointer is passed to that function as an argument
  - Passing large objects to functions without the overhead of copying them first
  - Accessing memory allocated on the heap
  - Referring to functions

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.

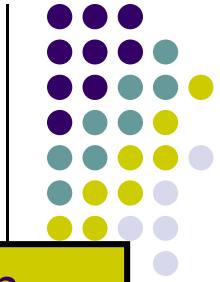
There are two general causes for these errors:
• Coding errors that end up setting the pointer to a strange number
• Use of a pointer that was at one time valid, but later became invalid

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 = 12;  /* 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.

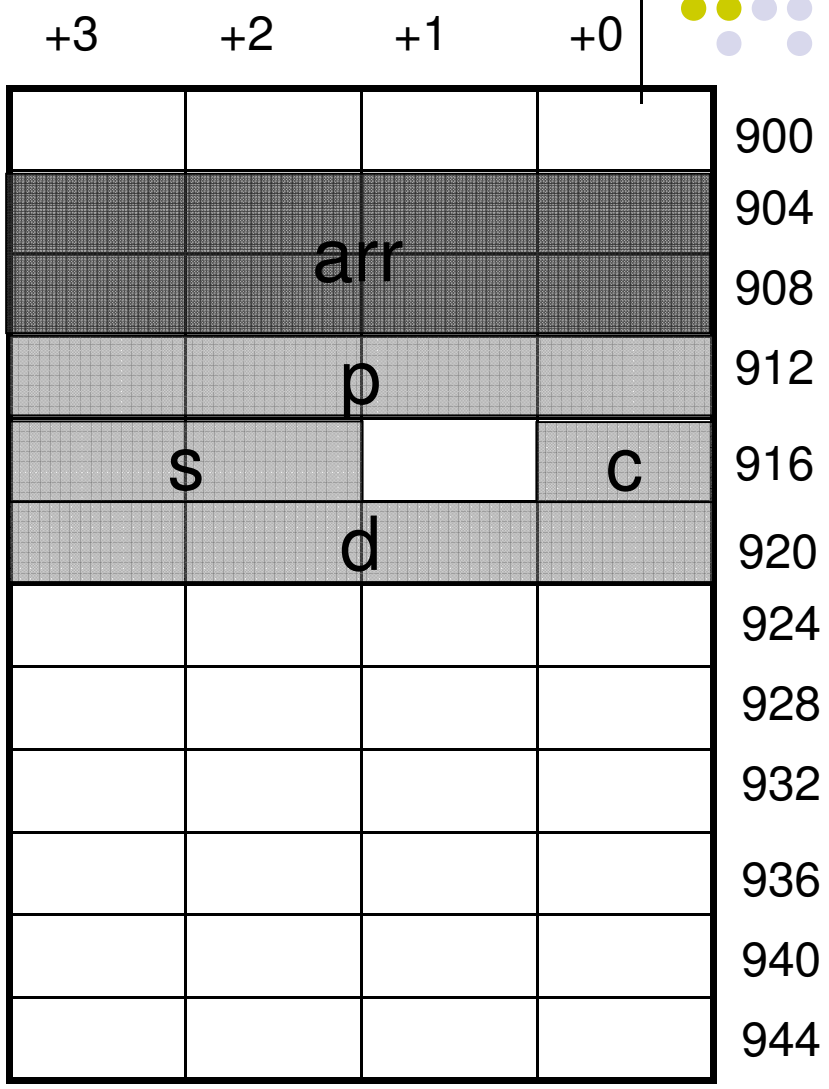
```
char * get_pointer()
{
  char x=0;
  return &x;
}

int main()
{
  char * ptr = get_pointer();
  *ptr = 12;  /* valid? */
  other_function();
  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 DevStudio when compiling:
main.cpp(6) : 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;
}
```

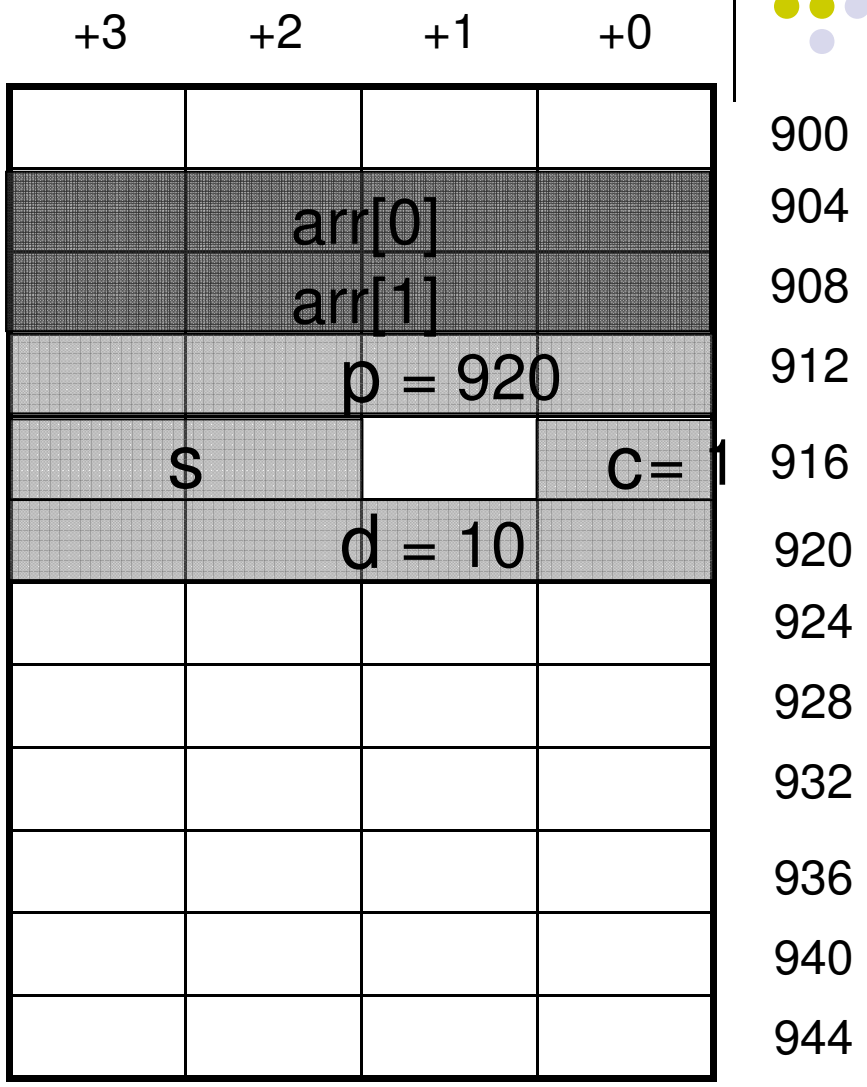


• NOTE: Here &d = 920 (in practice a 4-byte hex number such as 0x22FC3A08)

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

Example:
Usage of Pointers & Pointer Arithmetic

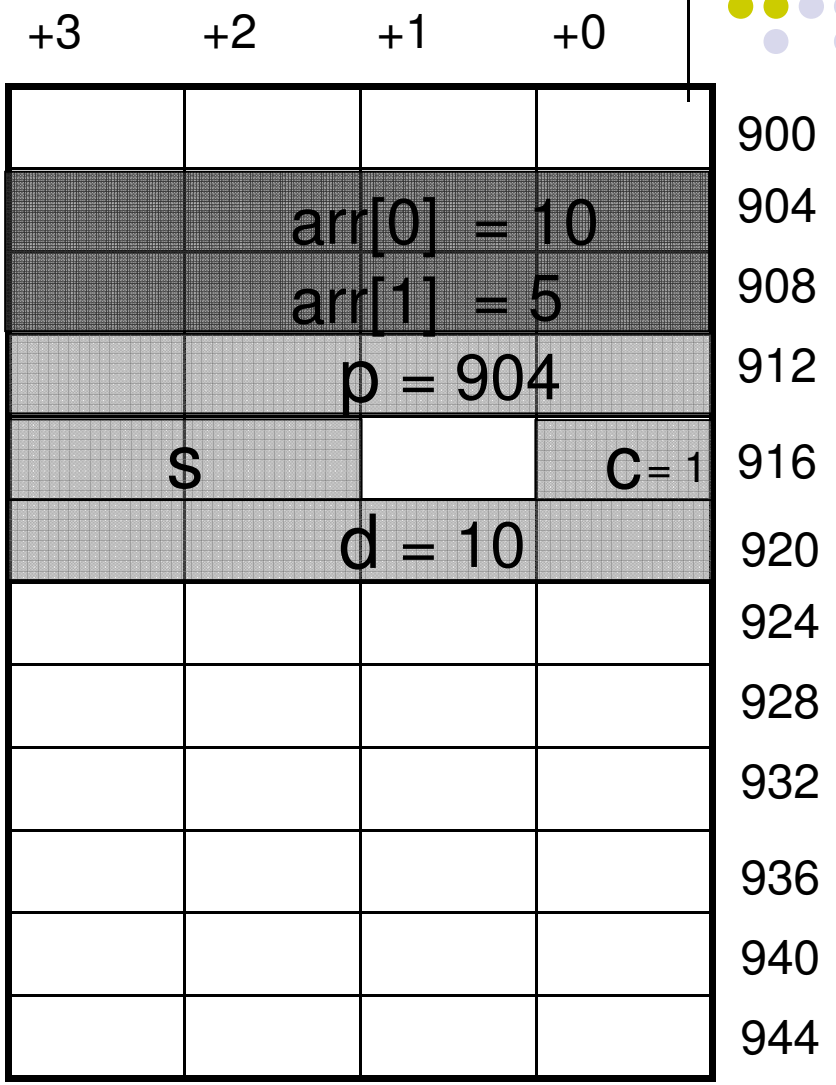
```
int main() {
      int d;
      char c;
      short s;
      int* p;
      int arr[2];

      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]

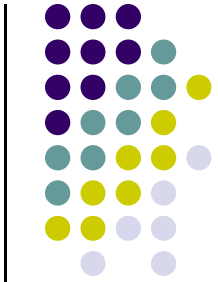
Example [Cntd.]

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

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

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



Question: arr[0] = ?

Use of pointers, another example…



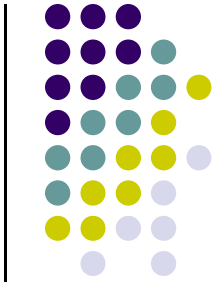
- Pass pointer parameters into function

```
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);
```

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.