Last Week

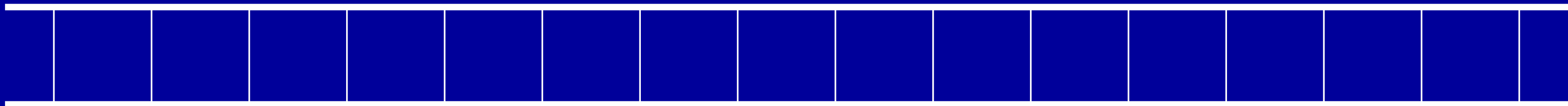
- Very short history of C
- Overview of the differences between C and Java
- The C language (keywords, types, functions, etc.)
- Preprocessor, header files

Overview

- Memory, a Program's Perspective
- More on Variables
- Memory Management
- Simple Pointers, Pointer Types and Casting
- Pointers and Arrays
- Function Pointers

Memory, a Program's Perspective (1)

- To a C program, memory is just a row of bytes
- ...



Memory, a Program's Perspective (1)

- To a C program, memory is just a row of bytes
- Each byte has some value, ...

	123	2	45	254	2	66	67	234	99	1	0	0	12	92	15	

Memory, a Program's Perspective (1)

- To a C program, memory is just a row of bytes
- Each byte has some value, and a location in the memory

123	2	45	254	2	66	67	234	99	1	0	0	12	92	15
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34

Memory, a Program's Perspective (2)

- When you define variables:

```
int count;
unsigned char c;
```

- ... two things happen:

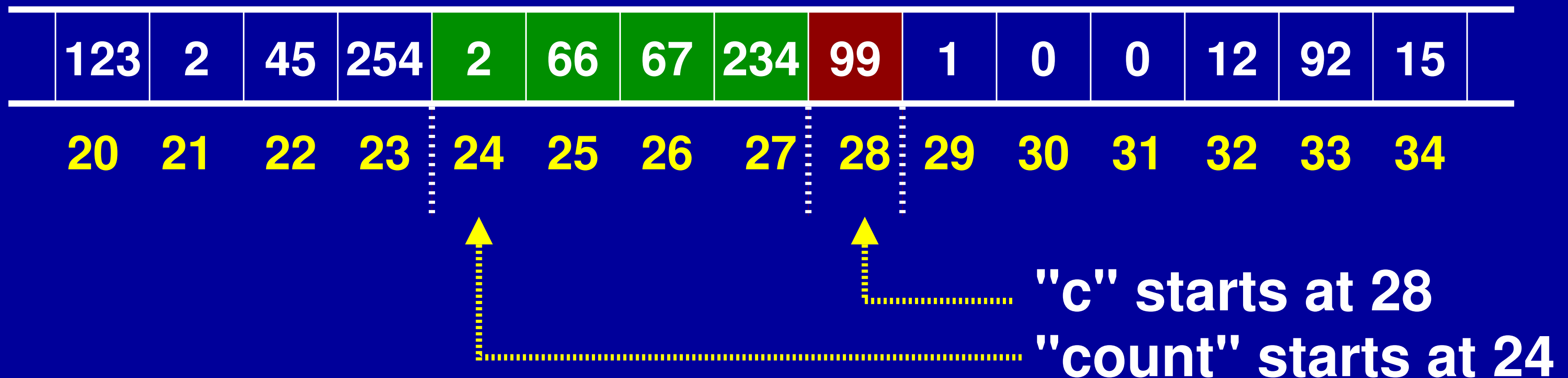
Memory, a Program's Perspective (3)

- Memory is reserved to store the variables
- ...

123	2	45	254	2	66	67	234	99	1	0	0	12	92	15
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34

Memory, a Program's Perspective (3)

- Memory is reserved to store the variables
- And the compiler 'remembers their location'



More on variables

- As a result, each variable has two values:
  - ⋆ The 'value' stored in the variable
  - ⋆ The 'location' of the memory used to store this value
- The variable name is simply 'shorthand' for the location
- The location is also called 'the address' of the variable

Pointers (1)

- A **pointer** is a variable that contains an address as it's value; it 'points to something'
  - ⋆ Pointers have the type **pointer to ...**
  - ⋆ Pointers can be declared using the ***** character

```
int *ptr;                  /* Pointer to int */
unsigned char *ch;         /* Pointer to unsigned char */
struct ComplexNumber *c;   /* Pointer to struct ComplexNumber */
int **pp;                  /* Pointer to pointer to int */
```

Pointers (2)

- An address of a variable can be obtained using an **&**
- The address returned is a **pointer to *type***

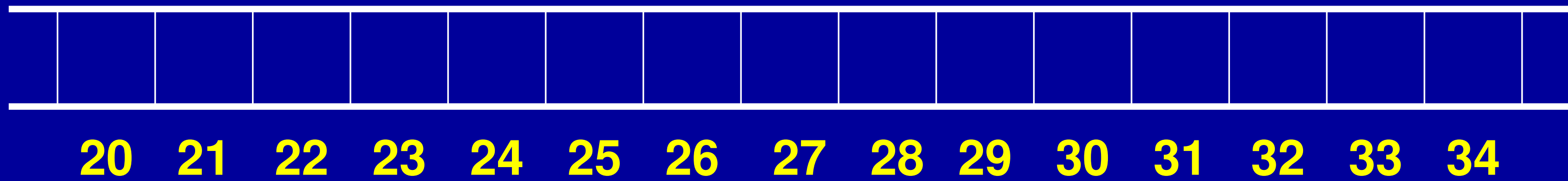
```
int i = 8;
int *p = &i;        /* OK, &i returns int *
                       p now points to i */
double *d = &i; /* ERROR, wrong pointer type */
```

Pointers (3)

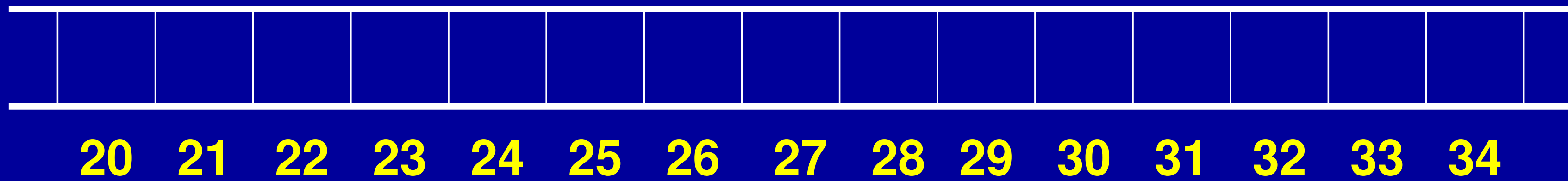
- A pointer can be **chased** using a *
- Returns **the value** in the location that is pointed to
- Can be use to write **a value** in the location

```
int i = 8;
int *p = &i;        /* p now points to i */
int j = *p;         /* j now cointains 8 */
*p = 12;            /* i now contains 12 */
```

Pointers (4)

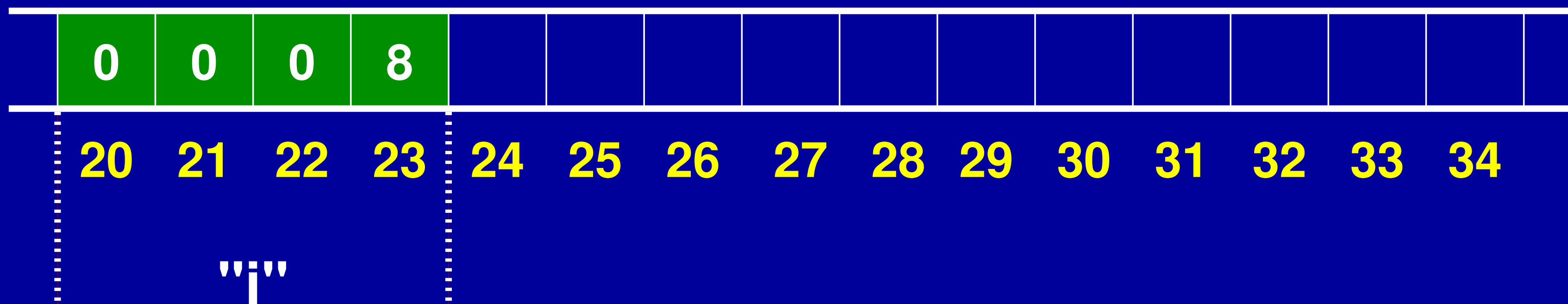


Pointers (4)



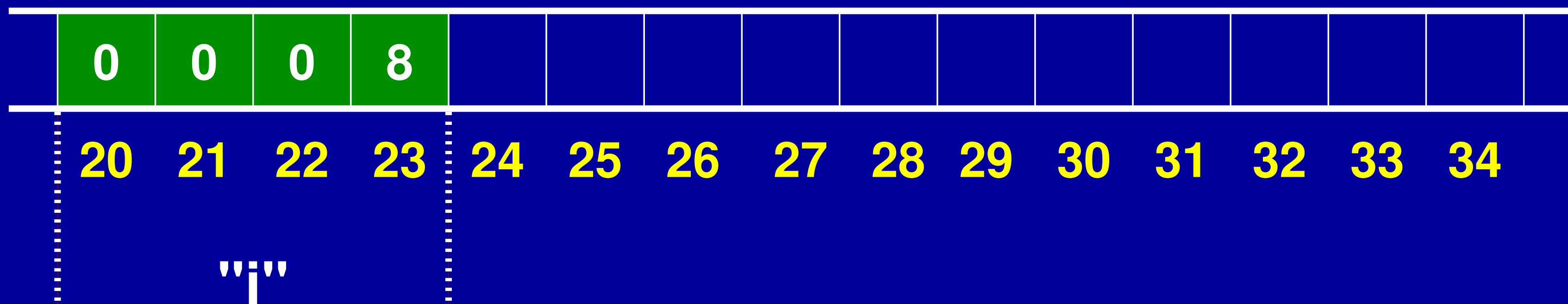
int i = 8

Pointers (4)



int i = 8
&i = 20

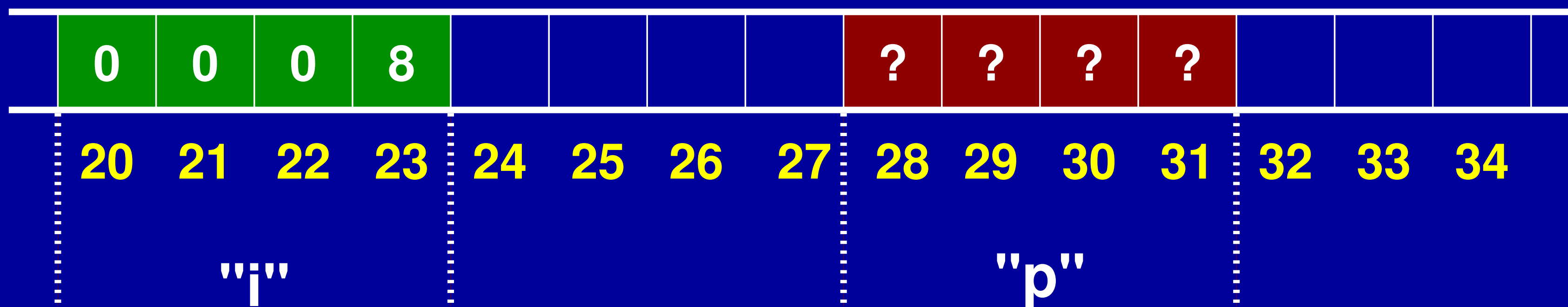
Pointers (4)



int i = 8
&i = 20

int *p

Pointers (4)



int i = 8
&i = 20

int *p
&p = 28

Pointers (4)

int i = 8
&i = 20

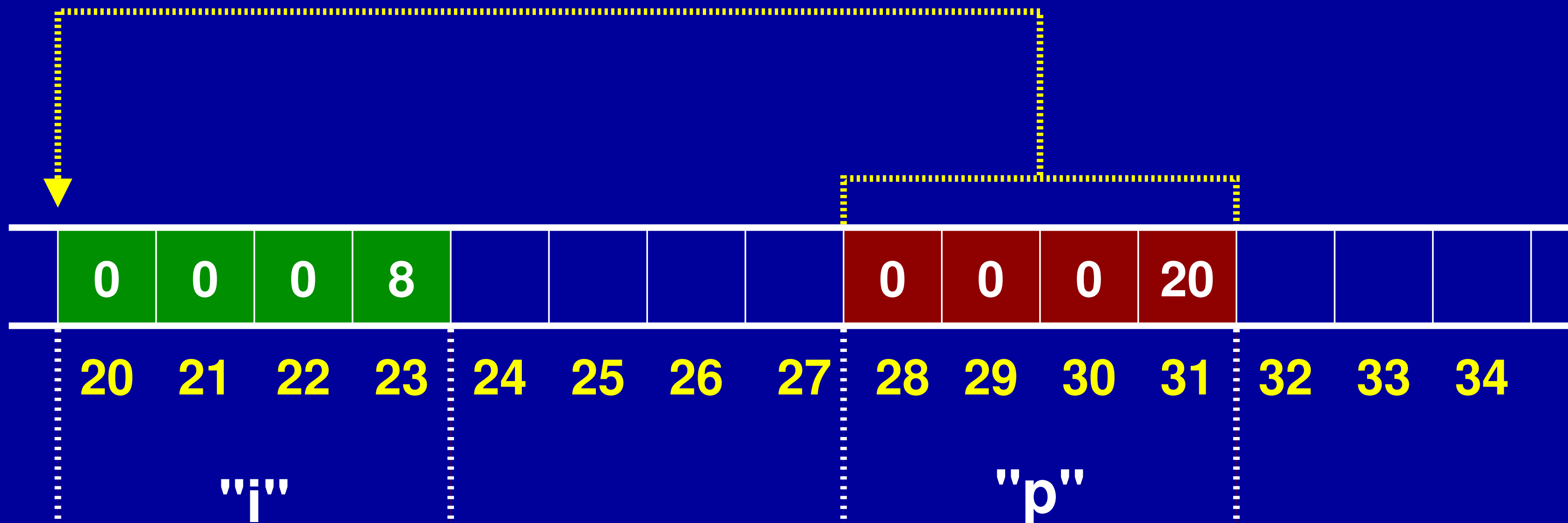
int *p = &i
&p = 28

Pointers (4)

int i = 8
&i = 20

int *p = &i (=20)
&p = 28

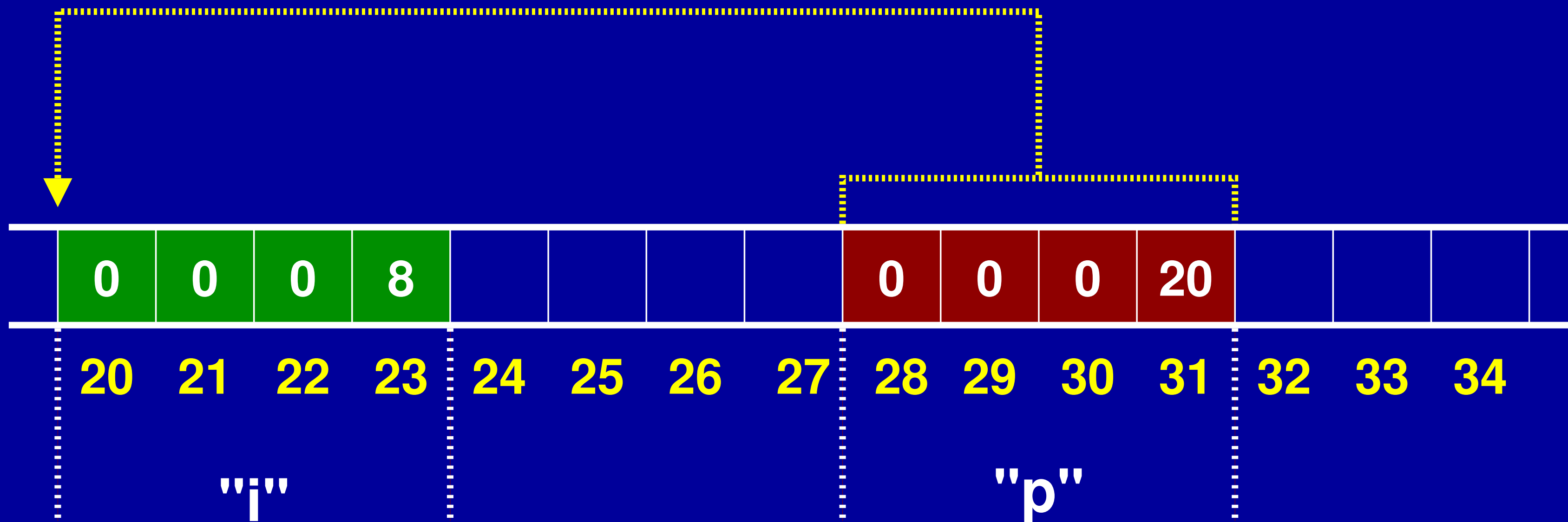
Pointers (4)



int i = 8
&i = 20

int *p = &i (=20)
&p = 28

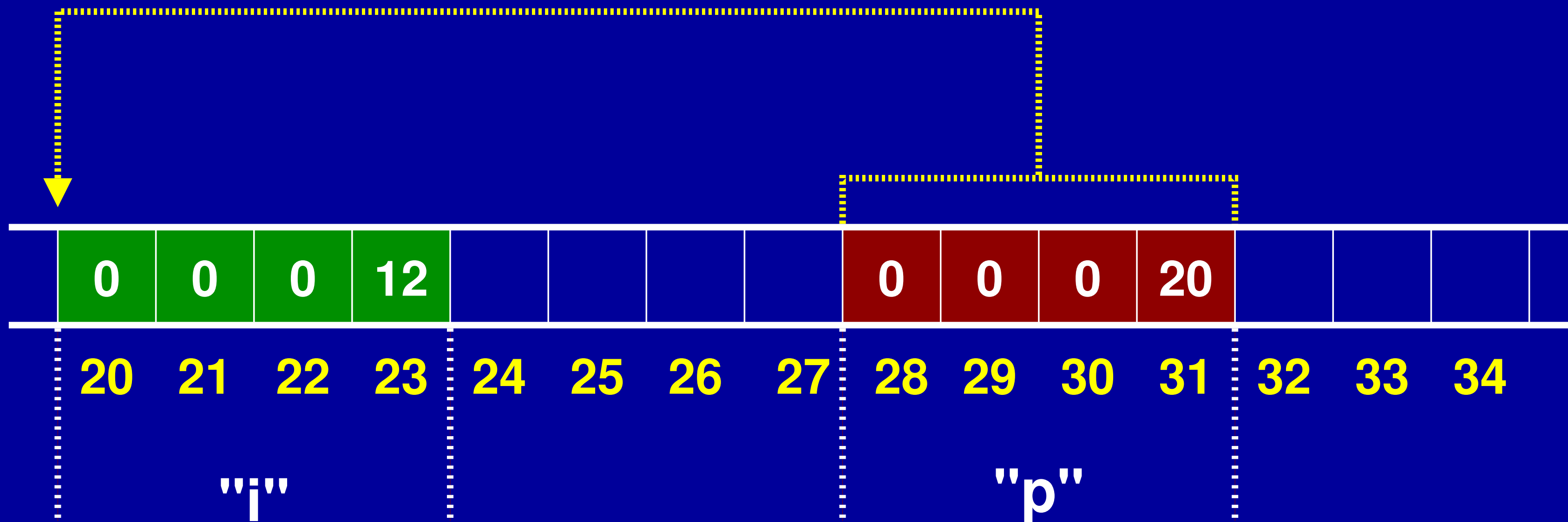
Pointers (4)



int i = 8
&i = 20

int *p = &i (=20)
&p = 28
int j = *p (= 8)

Pointers (4)



int i = 8
&i = 20

int *p = &i (=20)
&p = 28
int j = *p (= 8)

*p = 12

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
#include <stdio.h>

void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main(void) {
  int x = 9;
  int y = 5;

  swap(x, y);
  printf("x = %d, y = %d\n", x, y);
  return 0;
}
```

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
   #include <stdio.h>

>  void swap(int x, int y) {
     int temp = x;
     x = y;
     y = temp;
   }

   int main(void) {
     int x = 9;
     int y = 5;

     swap(x, y);
     printf("x = %d, y = %d\n", x, y);
     return 0;
   }
```

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
  #include <stdio.h>

> void swap(int *x, int *y) {
    int temp = x;
    x = y;
    y = temp;
  }

  int main(void) {
    int x = 9;
    int y = 5;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
  }
```

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
   #include <stdio.h>

>  void swap(int *x, int *y) {
>    int temp = *x;
     x = y;
     y = temp;
   }

   int main(void) {
     int x = 9;
     int y = 5;

     swap(x, y);
     printf("x = %d, y = %d\n", x, y);
     return 0;
   }
```

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
    #include <stdio.h>

>   void swap(int *x, int *y) {
>     int temp = *x;
>     *x = *y;
      y = temp;
    }

    int main(void) {
      int x = 9;
      int y = 5;

      swap(x, y);
      printf("x = %d, y = %d\n", x, y);
      return 0;
    }
```

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
   #include <stdio.h>

>  void swap(int *x, int *y) {
>    int temp = *x;
>    *x = *y;
>    *y = temp;
   }

   int main(void) {
     int x = 9;
     int y = 5;

     swap(x, y);
     printf("x = %d, y = %d\n", x, y);
     return 0;
   }
```

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
  #include <stdio.h>

> void swap(int *x, int *y) {
>   int temp = *x;
>   *x = *y;
>   *y = temp;
  }

  int main(void) {
    int x = 9;
    int y = 5;

>   swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
  }
```

Pointers (5)

- Pointers can be used to pass parameters **by reference**

```
#include <stdio.h>

void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(void) {
  int x = 9;
  int y = 5;

  swap(&x, &y);
  printf("x = %d, y = %d\n", x, y);
  return 0;
}
```

Pointers (6)

- Also useful for structs !

```
#include <stdio.h>

struct data {
   int counter;
   double value;
};
void add(struct data d, double value) {
   d.counter++;
   d.value += value;
}
int main(void) {
  struct data d = { 0, 0.0 };
  add(d, 1.0);
  add(d, 3.0);
  printf("counter = %d, value = %f\n", d.counter, d.value);
  return 0;
}
```

Pointers (7)

- Also useful for structs !

```
    #include <stdio.h>

    struct data {
        int counter;
        double value;
    };
>   void add(struct data *d, double value) {
>       (*d).counter++;
>       (*d).value += value;
    }
    int main(void) {
      struct data d = { 0, 0.0 };
>     add(&d, 1.0);
>     add(&d, 3.0);
      printf("counter = %d, value = %f\n", d.counter, d.value);
      return 0;
    }
```

Pointers (7)

- Also useful for structs !

```
#include <stdio.h>

struct data {
    int counter;
    double value;
};
void add(struct data *d, double value) {
>   d->counter++;
>   d->value += value;
}
int main(void) {
  struct data d = { 0, 0.0 };
  add(&d, 1.0);
  add(&d, 3.0);
  printf("counter = %d, value = %f\n", d.counter, d.value);
  return 0;
}
```

Arrays and Pointers (1)

- Special relationsship between arrays and pointers
- The C specifications says that

  `&array[0]`

  is equivalent to

  `array`

Arrays and Pointers (2)

- So the following statements are equivalent:

```
pointer = &array[0]
pointer = array
```

- So an array is simply **a pointer to the first element!**
- But it is constant however !

```
array = pointer; /* ERROR */
```

Arrays and Pointers (3)

- You can use pointers instead of arrays as parameters

```
#include <stdio.h>
void func1(int p[], int size) { }
void func2(int *p, int size) { }
int main(void) {
  int array[5];
  func1(array, 5);
  func2(array, 5);
  return 0;
}
```

Arrays and Pointers (4)

- You even use array indexing on pointers !

```
#include <stdio.h>

void clear(int *p, int size) {
    int i;
    for (i=0;i<size;i++) {
        p[i] = 0;
    }
}

int main(void) {
  int array[5];
  clear(array, 5);
  return 0;
}
```

Pointer Arithmetic (1)

- You can do calculations on pointers (pointer arithmetic)
- The **semantics depend on the size of a type!**
- You can determine the size of a type using **sizeof**

```
sizeof(int)                    /* returns 4 */
sizeof(double)                 /* returns 8 */
sizeof(struct ComplexNumber)   /* returns 16 */
```

Pointer Arithmetic (2)

- Pointer arithmetic works with the **size** of types!

```
int i = 8;
int *p = &i;
p++;  /* increases p with sizeof(int) */

struct ComplexNumber c = { 1.5, 0.5 };
struct ComplexNumber *cp = &c;
cp++; /* increases p with sizeof(struct ComplexNumber) */
```

Pointer Arithmetic (3)

- This is obvious when using pointers as arrays:

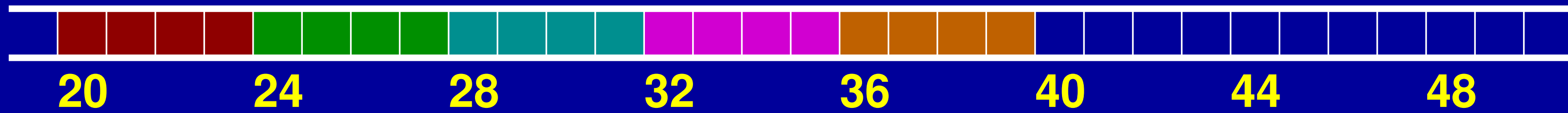
```
int i;
int array[5];
int *p = array;

for (i=0;i<5;i++) {
    *p = 0;
    p++;
}
```

Pointer Arithmetic (4)

Pointer Arithmetic (4)

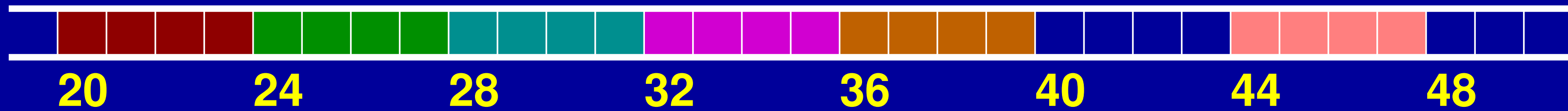
```
int array[5];
```

Pointer Arithmetic (4)



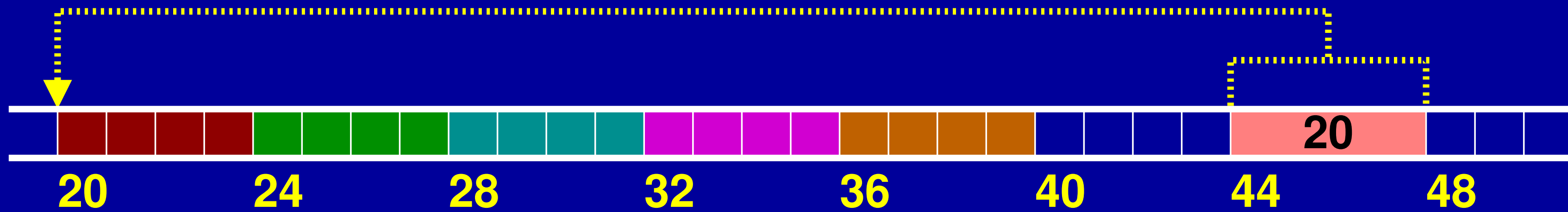
int array[5];

Pointer Arithmetic (4)



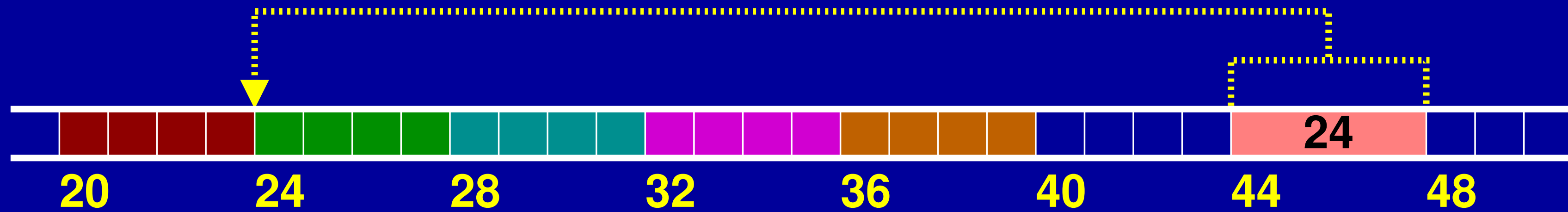

```
int array[5];
int *p
```

Pointer Arithmetic (4)




```
int array[5];
int *p = array;
```

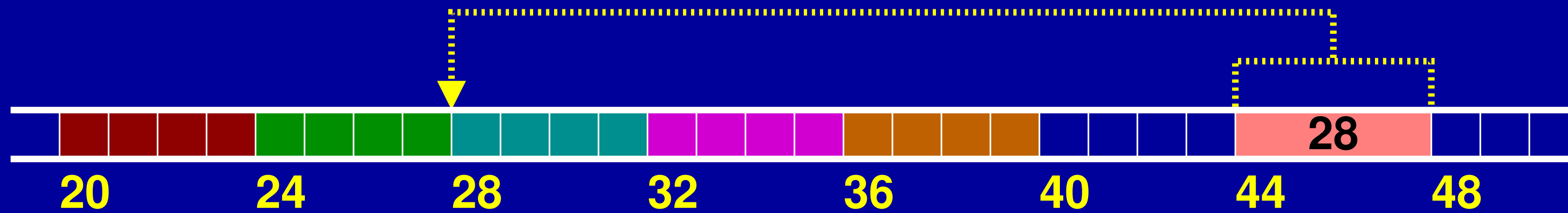
Pointer Arithmetic (4)




```
int array[5];
int *p = array;
```

```
p++;
```

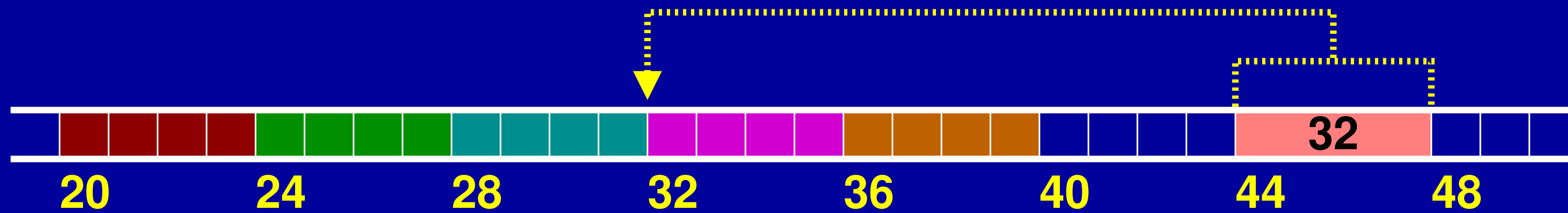
Pointer Arithmetic (4)




```
int array[5];
int *p = array;
```

```
p++;
p++;
```

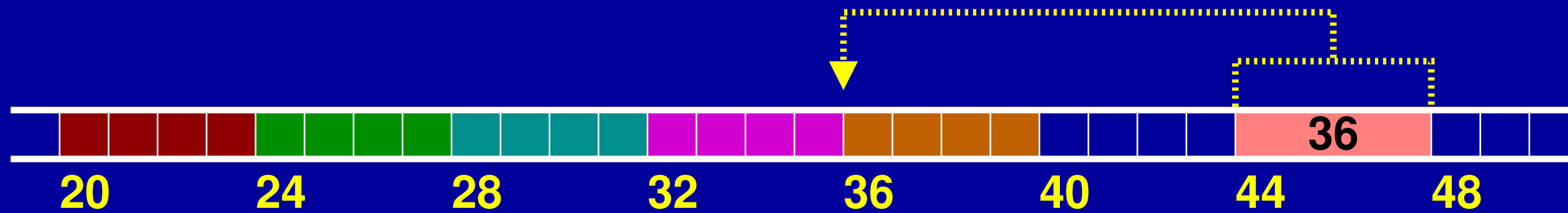
Pointer Arithmetic (4)




```
int array[5];
int *p = array;
```

```
p++; p++;
p++;
```

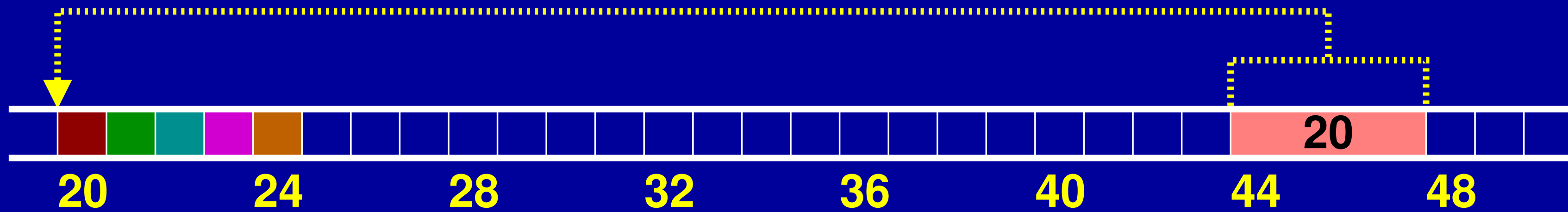
Pointer Arithmetic (4)




```
int array[5];
int *p = array;
```

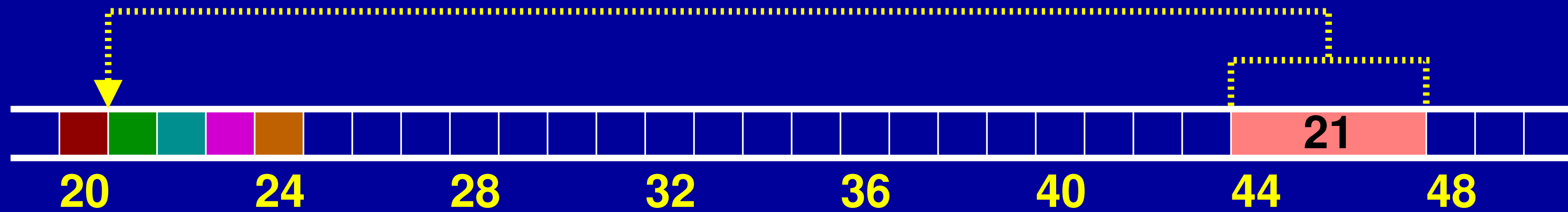
```
p++; p++;
p++; p++;
```

Pointer Arithmetic (4)




```
char array[5];
char *p = array;
```

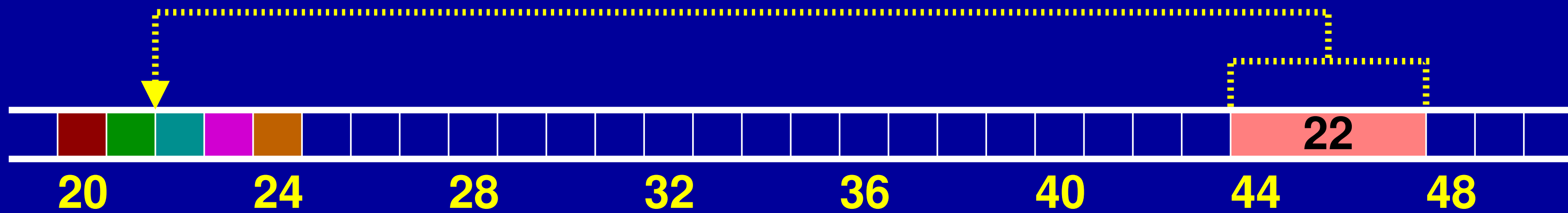
Pointer Arithmetic (4)




```
char array[5];
char *p = array;
```

```
p++;
```

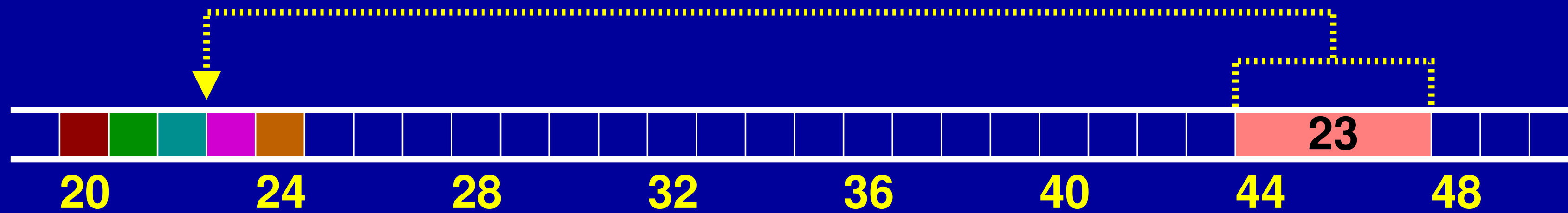
Pointer Arithmetic (4)




```
char array[5];
char *p = array;
```

```
p++;
p++;
```

Pointer Arithmetic (4)




```
char array[5];

char *p = array;
```

```
p++; p++;
p++;
```

Arrays and Pointers (5)

- An array is really a pointer, so array indexing is pointer arithmetic

  `array[i]`

  is equivalent to

  `*(array+i)`

Arrays and Pointers (6)

- This allows efficient implementations (e.g., *strcpy*):

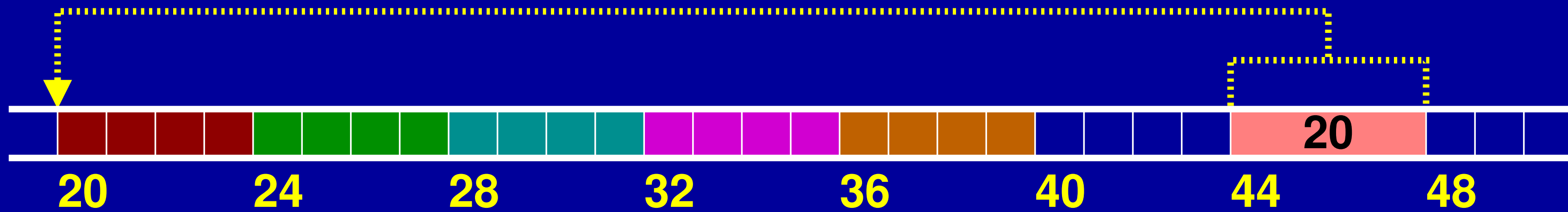
```
void copy(char to[], char from[]) {
    int i = 0;
    while (from[i] != '\0') {
        to[i] = from[i];
        i++;
    }
    to[i] = '\0';
}

void copy(char *to, char *from) {
     while (*from != '\0') {
         *to++ = *from++;
     }
     *to = '\0';
}
```

Pointer Arithmetic (5)

- It is very important that the pointer type is correct!

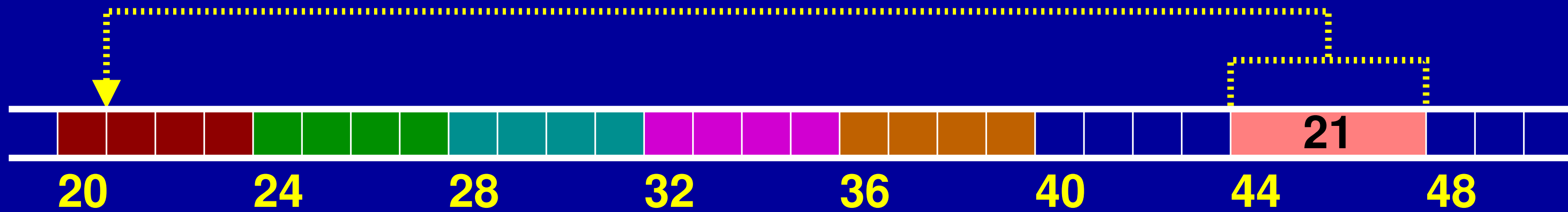
```
int array[5];
char *p = (char *)array;
p++; /* increases p with sizeof(char) */
int x = *p;
```

Pointer Arithmetic (6)




```
int array[5];
char *p = (char *) array;
```

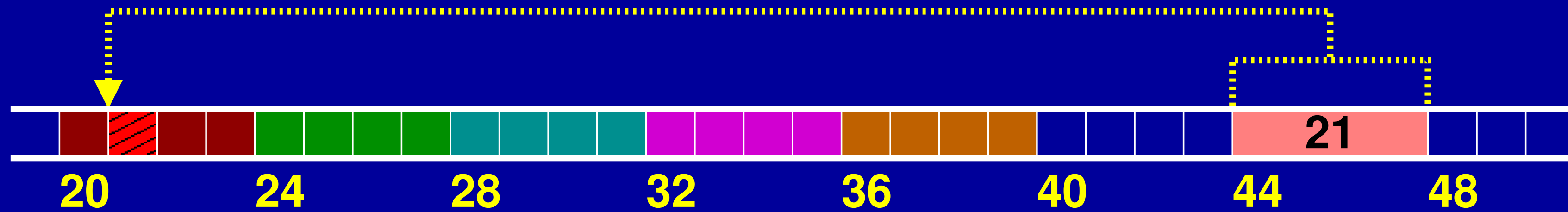
Pointer Arithmetic (6)




```
int array[5];
char *p = (char *) array;
```

```
p++;
```

Pointer Arithmetic (6)




```
int array[5];
char *p = (char *) array;
```

```
p++;
int x = *p;
```

Memory Management (1)

- Until now, all data has been static
  - ⋆ Memory reserved at compile time
- You can also dynamically use memory at runtime
  - ⋆ `malloc` reserves memory
  - ⋆ `free` releases memory
  - ⋆ exported by `stdlib.h`

Memory Management (2)

- Prototypes:

```
void *malloc(size_t size);
void free(void *ptr);
```

- Both use **void ***, a pointer without a type!
- You can assign this to any other pointer type
- Do **NOT** directly use **void ***

Memory Management (3)

```
#include <stdlib.h>

int main(void) {
    int i;

    int *arr = malloc(10*sizeof(int));

    for (i=0;i<10;i++) {
        arr[i] = 7;
    }

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

Memory Management (4)

- Unlike arrays, dynamically allocated memory can be returned from a function.

```
#include <stdlib.h>

int *createIntArray(int size) {
    return malloc(size*sizeof(int));
}

int main(void) {
    int *arr = createIntArray(10);
    free(arr);
    return 0;
}
```

Memory Management (5)

- You must **always** keep a pointer to allocated memory
- You need this to use it, and free it later
- If you don't, you've got a **memory leak**
- Memory leaks will slowly reserve all the machine memory, causing the program (or the machine) to crash eventually!

Memory Management (6)

```
#include <stdlib.h>
int main(void) {
    while (1) {
        malloc(10); /* LEAK!! */
    }
    return 0;
}
```

Memory Management (7)

- If you run out of memory, `malloc` will return `NULL`

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

int main(void) {

    int *array = malloc(10*sizeof(int));

    if (array == NULL) {
        printf("Out of memory!\n");
        exit(1);
    }

    /* do something useful here */
    return 0;
}
```

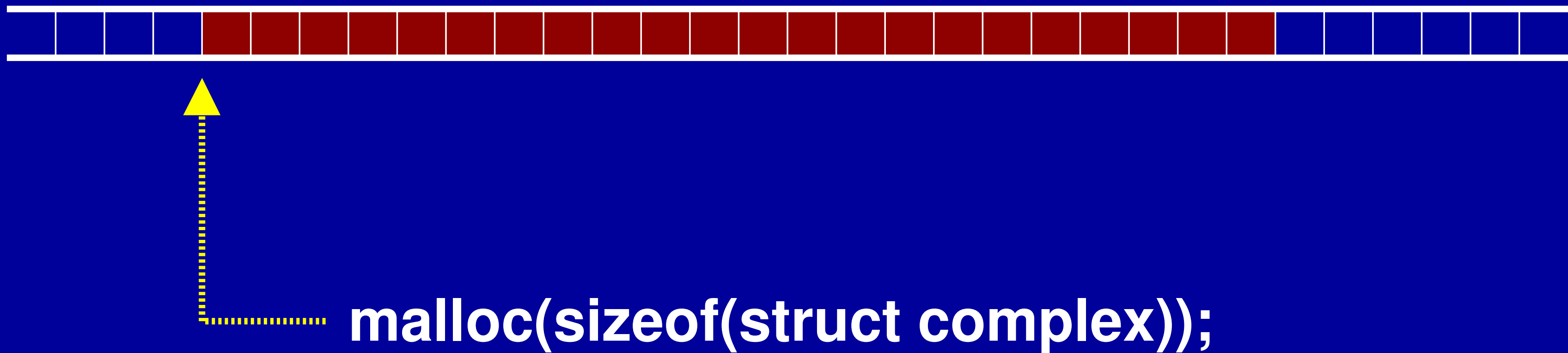
Memory Management (8)

- What happens if you mix malloc and structs ?

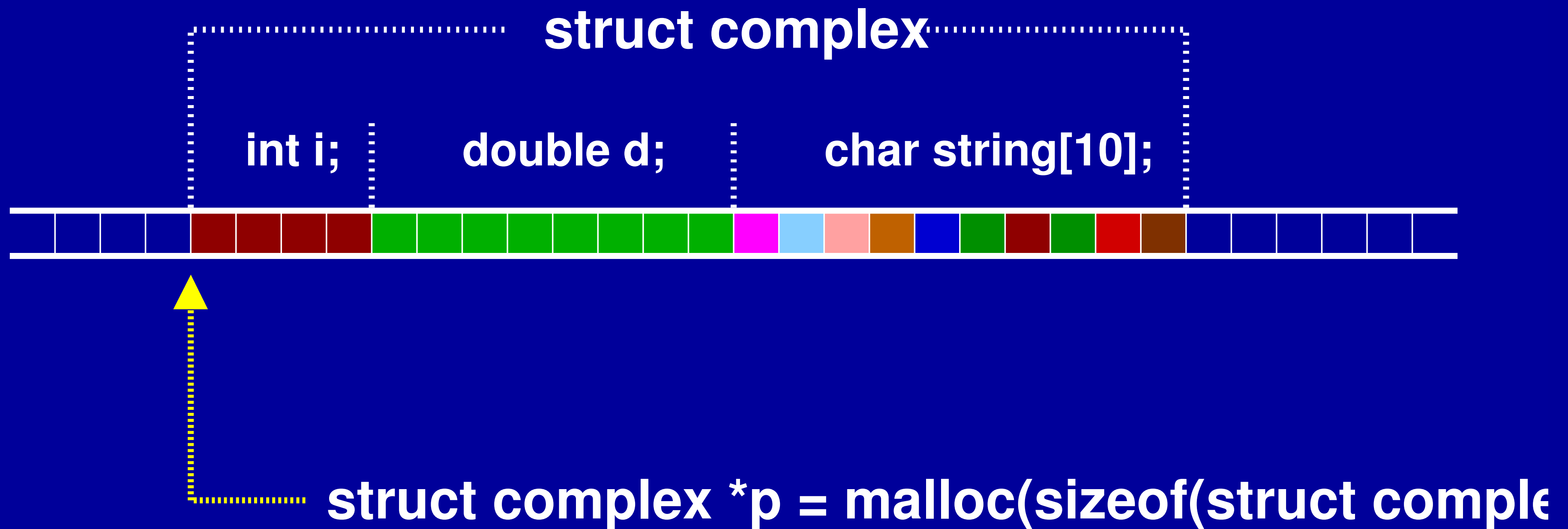
```
#include <stdlib.h>

struct complex {
    int i;
    double d;
    char string[10];
}

int main(void) {
    struct complex *c = malloc(sizeof(struct complex));

    /* do something useful here */
    return 0;
}
```

Memory Management (9)

Memory Management (9)



Memory Management (9)



Multidimensional Arrays (1)

- Static multidimensional arrays can be declared like this:

  `short array[`*ROW*`][`*COL*`];`

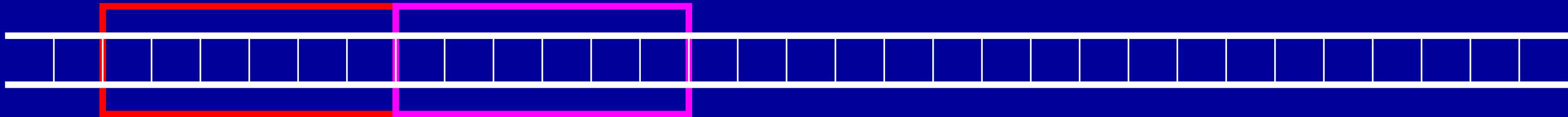
- The memory is reserved at compile time
- The compiler needs to know the exact size
- Unlike Java, this is a single block of memory, **NOT** an array of arrays !!!

Multidimensional Arrays (2)

Multidimensional Arrays (2)

```
short array[2][3];
```

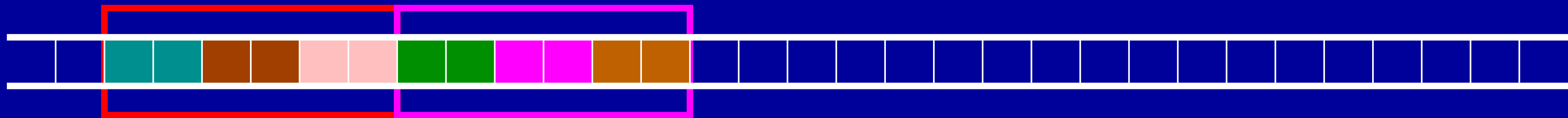
Multidimensional Arrays (2)



short array[2][3];

– create array[2] of element short[3]

Multidimensional Arrays (2)



short array[2][3];

– create array[2] of element short[3]

Multidimensional Arrays (2)

short array[2][3] == short array[6] !!!

Multidimensional Arrays (3)

- How can you malloc multidimensional arrays ?
  - ⋆ malloc a block of data and convince C its a multidimensional array
  - ⋆ malloc an array of pointers that point to arrays (like Java does)
- There is a **very** big difference between these approaches!!!

Multidimensional Arrays (4)

```
#define ROWS 2
#define COLS 3

int main(void) {

  int i,j;
  short (*array)[COLS] = malloc(ROWS*COLS*sizeof(short));

  for (i=0;i<ROWS;i++) {
     for (j=0;j<COLS;j++) {
        array[i][j] = 42;
     }
  }
  /* etc. */
}
```

Multidimensional Arrays (4)

```
  #define ROWS 2
  #define COLS 3

  int main(void) {

    int i,j;
> short (*array)[COLS] = malloc(ROWS*COLS*sizeof(short));

    for (i=0;i<ROWS;i++) {
       for (j=0;j<COLS;j++) {
>         a[i][j] = 42;
       }
    }
    /* etc. */
  }
```

Multidimensional Arrays (5)

Multidimensional Arrays (5)

short array[2][3] (or short array[6])

Multidimensional Arrays (5)

short array[2][3] (or short array[6])

short (*array)[3] = malloc(...)

Multidimensional Arrays (5)

short array[2][3] (or short array[6])

short (*array)[3] = malloc(...)

Multidimensional Arrays (6)

```
#define ROWS 2
#define COLS 3

int main(void) {

  int i,j;
  short **array = malloc(ROWS*sizeof(short *));

  for (i=0;i<ROWS;i++) {
     array[i] = malloc(COLS*sizeof(short));
  }

  for (i=0;i<ROWS;i++) {
     for (j=0;j<COLS;j++) {
        array[i][j] = 42;
     }
  }
  /* etc. */
}
```

Multidimensional Arrays (6)

```
#define ROWS 2
#define COLS 3

int main(void) {

  int i,j;
> short **array = malloc(ROWS*sizeof(short *));

  for (i=0;i<ROWS;i++) {
     array[i] = malloc(COLS*sizeof(short));
  }

  for (i=0;i<ROWS;i++) {
     for (j=0;j<COLS;j++) {
        array[i][j] = 42;
     }
  }
  /* etc. */
}
```

Multidimensional Arrays (6)

```
  #define ROWS 2
  #define COLS 3

  int main(void) {

     int i,j;
>    short **array = malloc(ROWS*sizeof(short *));

>    for (i=0;i<ROWS;i++) {
>       array[i] = malloc(COLS*sizeof(short));
>    }

     for (i=0;i<ROWS;i++) {
        for (j=0;j<COLS;j++) {
           array[i][j] = 42;
        }
     }
     /* etc. */
  }
```

Multidimensional Arrays (6)

```
  #define ROWS 2
  #define COLS 3

  int main(void) {

    int i,j;
>   short **array = malloc(ROWS*sizeof(short *));

>   for (i=0;i<ROWS;i++) {
>      array[i] = malloc(COLS*sizeof(short));
>   }

    for (i=0;i<ROWS;i++) {
       for (j=0;j<COLS;j++) {
>         array[i][j] = 42;
       }
    }
    /* etc. */
  }
```

Multidimensional Arrays (7)

short array[2][3] (or short array[6])

Multidimensional Arrays (7)

short array[2][3]

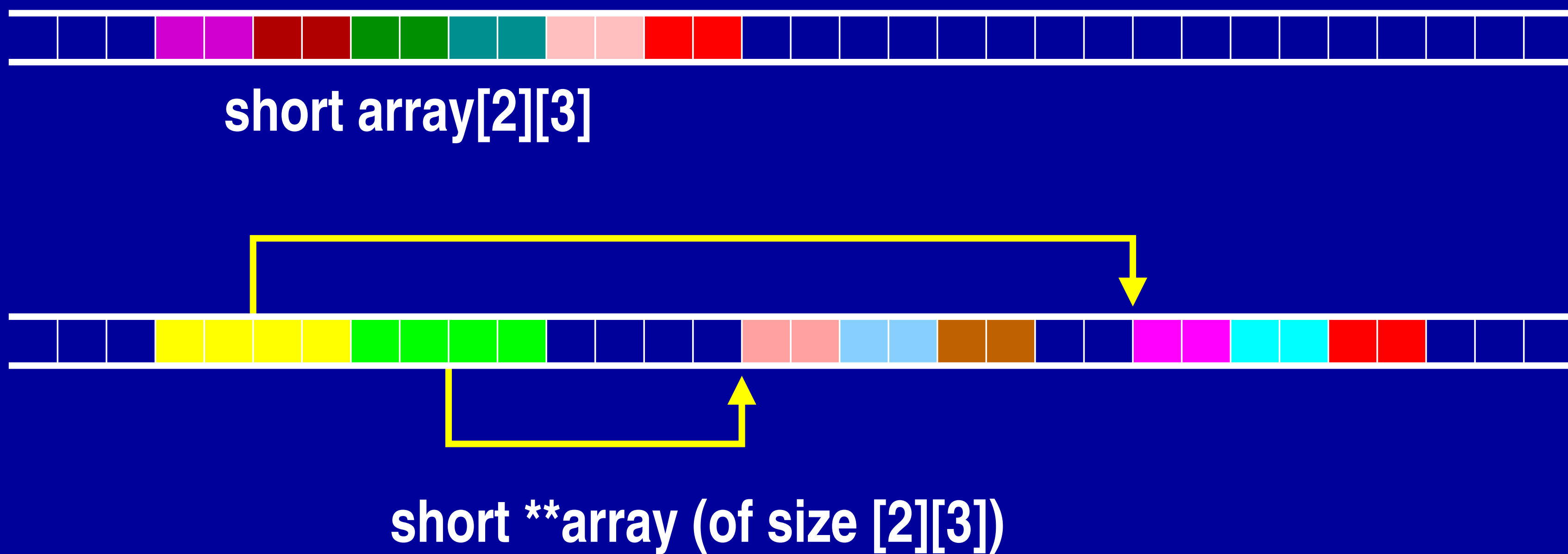
short **array (of size [2][3])

Multidimensional Arrays (7)

short array[2][3]

short **array (of size [2][3])

Multidimensional Arrays (7)



Multidimensional Arrays (8)

- Arrays of arrays:
  - ⋆ (-) contains more data!!
  - ⋆ (-) do not have to contiguous in memory

  - ⋆ (+) rows may be of different length
  - ⋆ (+) rows can be swapped or replaced
- The're very similar to Java's arrays of arrays!

Function Pointers (1)

- C also supports pointers to functions
- They have the somewhat complex syntax:

```
type (*identifier) (parameter-list);
```

Function Pointers (2)

- For example, a pointer that can refer to a function with two `int` parameters and a `double` result:

```
double (*func) (int p1, int p2);
```

- Note that `func` is the name of the pointer variable
- Function pointers can be used as variables, parameters, and return types

Function Pointers (3)

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b ? a : b);
}
int mul(int x, int y) {
    return x*y;
}

int main(void) {
    int result;
    int (*func) (int p1, int p2);

    func = max;
    result = func(1, 2);
    printf("%d\n", result);

    func = mul;
    result = func(3,4);
    printf("%d\n", result);
    return 0;
 }
```

Function Pointers (4)

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b ? a : b);
}

int foo(int a, int b, int (*func) (int p1, int p2)) {
    return func(a, b);
}

int main(void) {
     int result = foo(11, 22, max);
     printf("%d\n", result);
     return 0;
}
```

Function Pointers (5)

```
#include <stdio.h>

int max(int a, int b) {
    return (a > b ? a : b);
}

int (*faa(void)) (int p1, int p2) {
  return max;
}

int main(void) {
     int result;
     int (*func) (int p1, int p2);
     func = faa();
     result = func(11, 22);
     printf("%d\n", result);
     return 0;
}
```

Conclusion (1)

- Pointers are variable that contain an address
- Pointers can point to data or functions
- Pointers and arrays are closely related
- The type of the pointer used in pointer arithmetic

Conclusion (2)

- `malloc` and `free` can be used for memory allocation
- Always assign the result of `malloc` to a non-void pointer and check for `NULL`