CSE 421/521 - Operating Systems
Fall 2019 Recitations

RECITATION - I

RECAP C PROGRAMMING

PROF. TEVFIK KOSAR

Presented by Hanbin Zhang

University at Buffalo
September 9-13, 2019

Basic C Program: Print to stdout

```
#include <stdio.h>

main()
{
    printf("Hello, CSE421-521 Class!\n");
}

---
gcc prog1.c              ==> a.out
gcc prog1.c -o prog1     ==> prog1
make prog1               ==> prog1
```

Header Files

- The C compiler works in 3 phases:
  1. Pre-process source files
  2. Compile source files into object files
  3. Link object files into an executable
- `#include <stdio.h>` means “include the contents of standard file `stdio.h` here”
  1. Standard files are usually located in directory /usr/include
  2. /usr/include/stdio.h may contain #include statements itself...
- You can use #include to include your own files into each other:
  - `#include "myfile.h"` means: “include file myfile.h (from the current directory) here”
  - Included files usually have extension “.h” (header)

Basic Data Types

- Basic Types
  - char : character - 1 byte
  - short: short integer - 2 bytes
  - int: integer - 4 bytes
  - long: long integer - 4 bytes
  - float: floating point - 4 bytes
  - double - double precision floating point - 8 bytes
- Formatting Template
  - %d: integers
  - %f: floating point
  - %c: characters
  - %s: string
  - %x: hexadecimal
  - %u: unsigned int

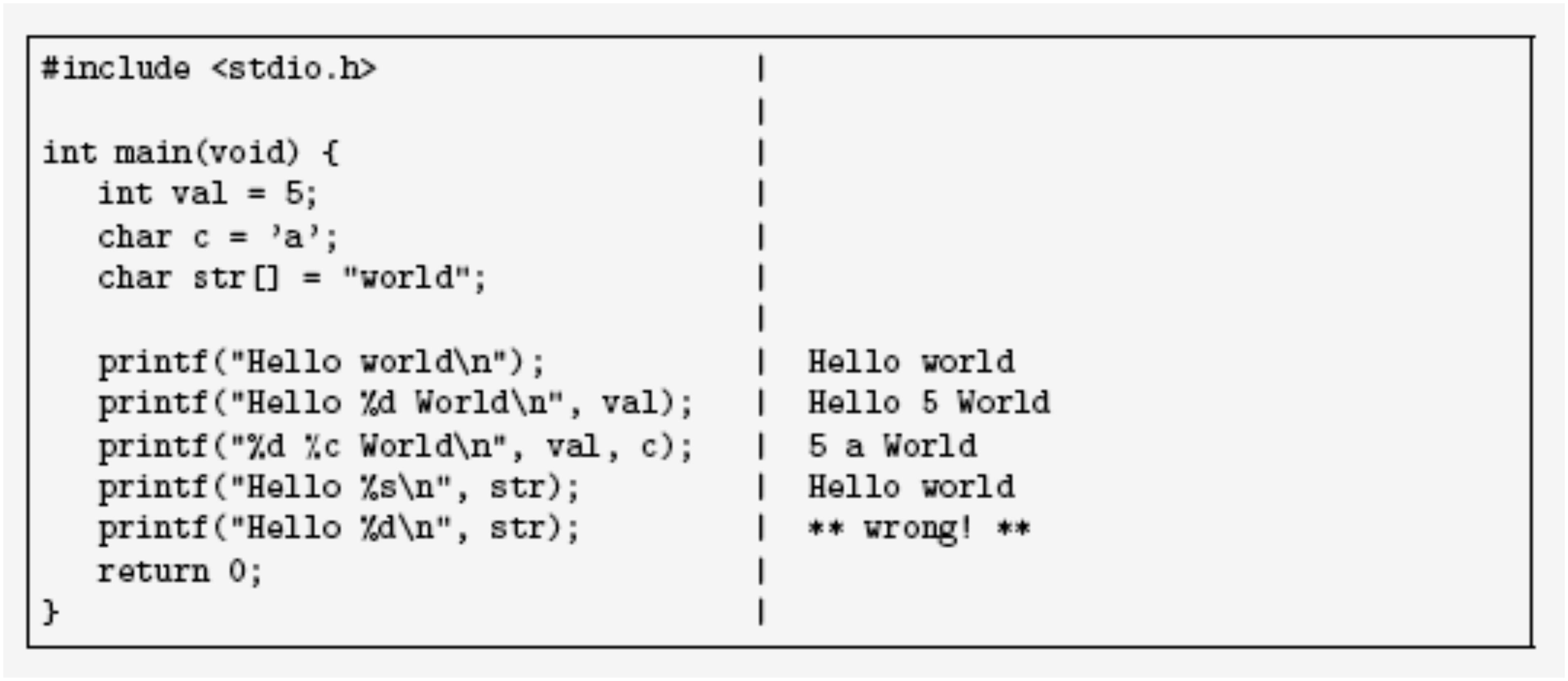
Test Size of Data Types

```
#include <stdio.h>

main()
{
    printf("sizeof(char): %d\n", sizeof(char));
    printf("sizeof(short): %d\n", sizeof(short));
    printf("sizeof(int): %d\n", sizeof(int));
    printf("sizeof(long): %d\n", sizeof(long));
    printf("sizeof(float): %d\n", sizeof(float));
    printf("sizeof(double): %d\n", sizeof(double));
}
```

Formatting

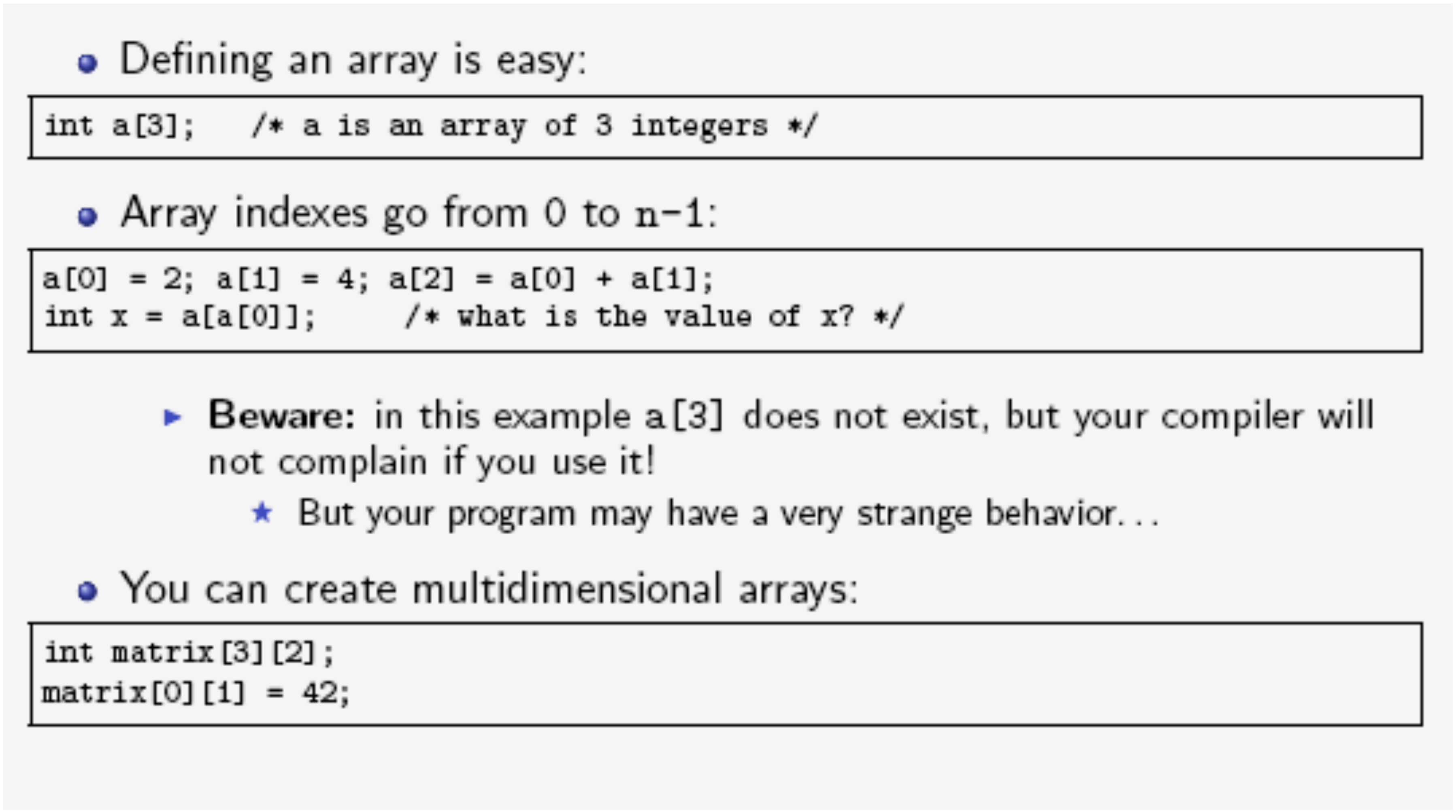
```
#include <stdio.h>

main()
{
	char var1;
	float f;

	printf(" Enter a character:");
	scanf("%c", &var1);
	printf("You have entered character:%c \n ASCII value=%d \n
			Address=%x\n", var1, var1, &var1);

	printf(" And its float value would be: %.2f\n", (float)var1);
}
```

Formatting *(cont.)*



```
#include <stdio.h>                      |
                                        |
int main(void) {                        |
   int val = 5;                         |
   char c = 'a';                        |
   char str[] = "world";                |
                                        |
   printf("Hello world\n");             |  Hello world
   printf("Hello %d World\n", val);     |  Hello 5 World
   printf("%d %c World\n", val, c);     |  5 a World
   printf("Hello %s\n", str);           |  Hello world
   printf("Hello %d\n", str);           |  ** wrong! **
   return 0;                            |
}                                       |
```

Arrays



- Defining an array is easy:

```
int a[3];   /* a is an array of 3 integers */
```

- Array indexes go from 0 to `n-1`:

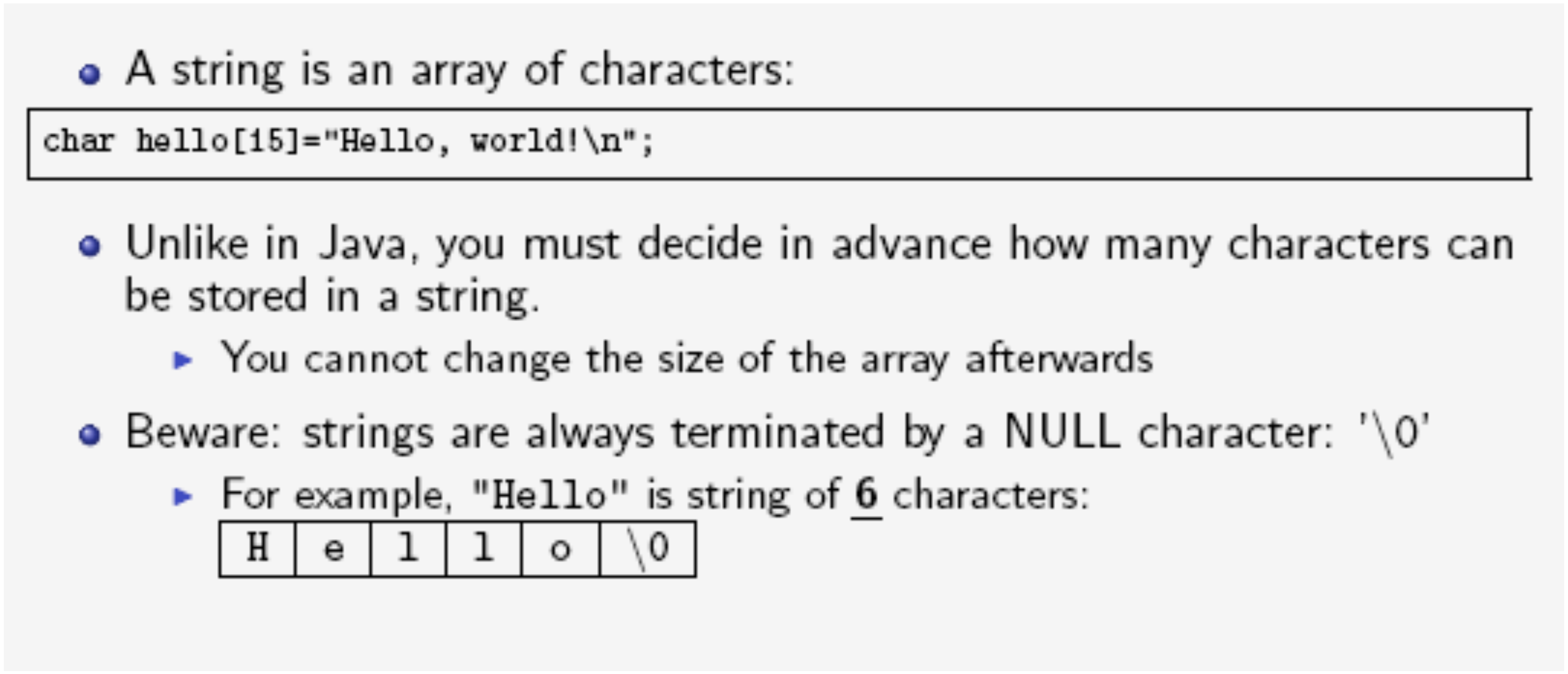
```
a[0] = 2; a[1] = 4; a[2] = a[0] + a[1];
int x = a[a[0]];     /* what is the value of x? */
```

  - **Beware:** in this example `a[3]` does not exist, but your compiler will not complain if you use it!
    - ★ But your program may have a very strange behavior. . .

- You can create multidimensional arrays:

```
int matrix[3][2];
matrix[0][1] = 42;
```

Strings



- A string is an array of characters:

```
char hello[15]="Hello, world!\n";
```

- Unlike in Java, you must decide in advance how many characters can be stored in a string.
  - You cannot change the size of the array afterwards
- Beware: strings are always terminated by a NULL character: '\0'
  - For example, "`Hello`" is string of <u>**6**</u> characters:

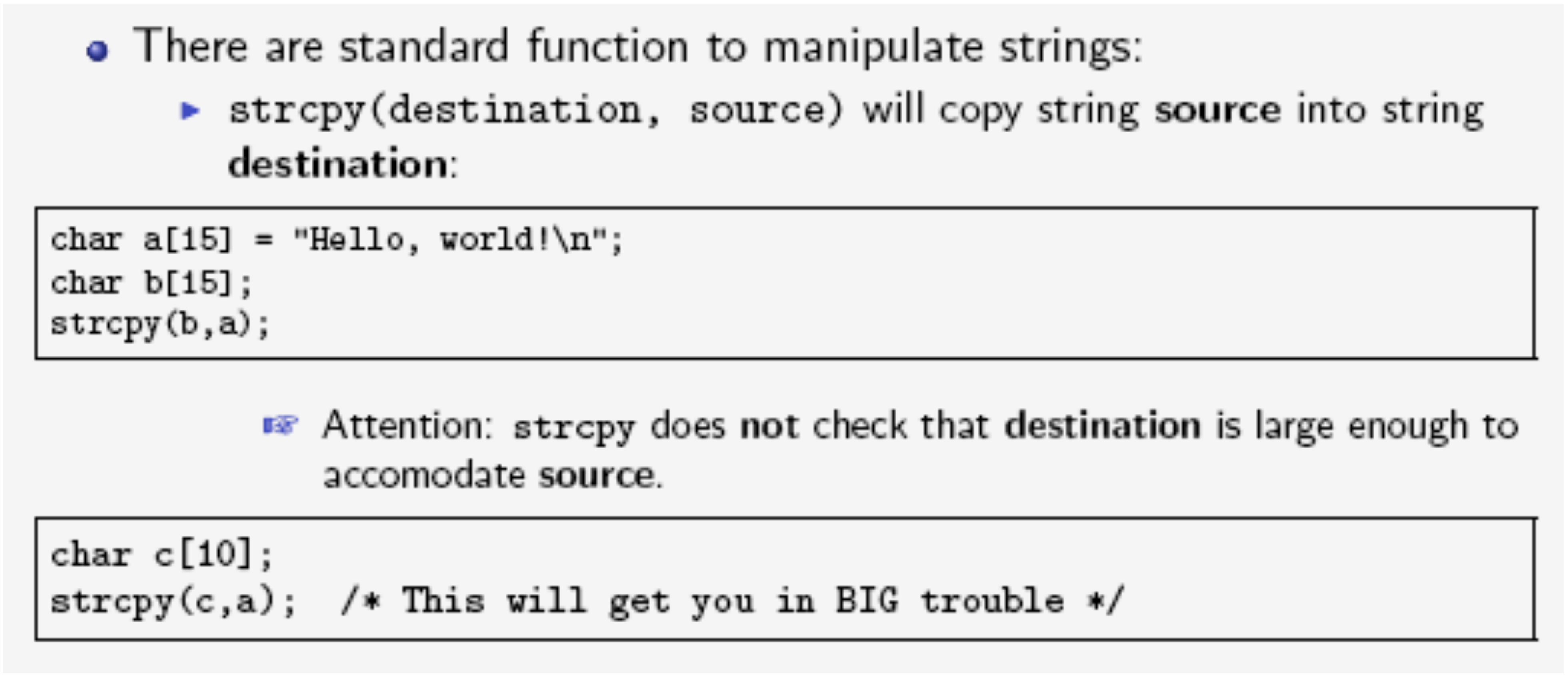
H	e	l	l	o	\0

Manipulating Arrays

- You cannot copy an array into another directly
    - You must copy each element one at a time

```
int a[3] = {12,24,36};
int b[3];

b = a;      /* This will NOT work! */

b[0]=a[0];
b[1]=a[1];
b[2]=a[2]; /* This will work */
```

Manipulating Strings



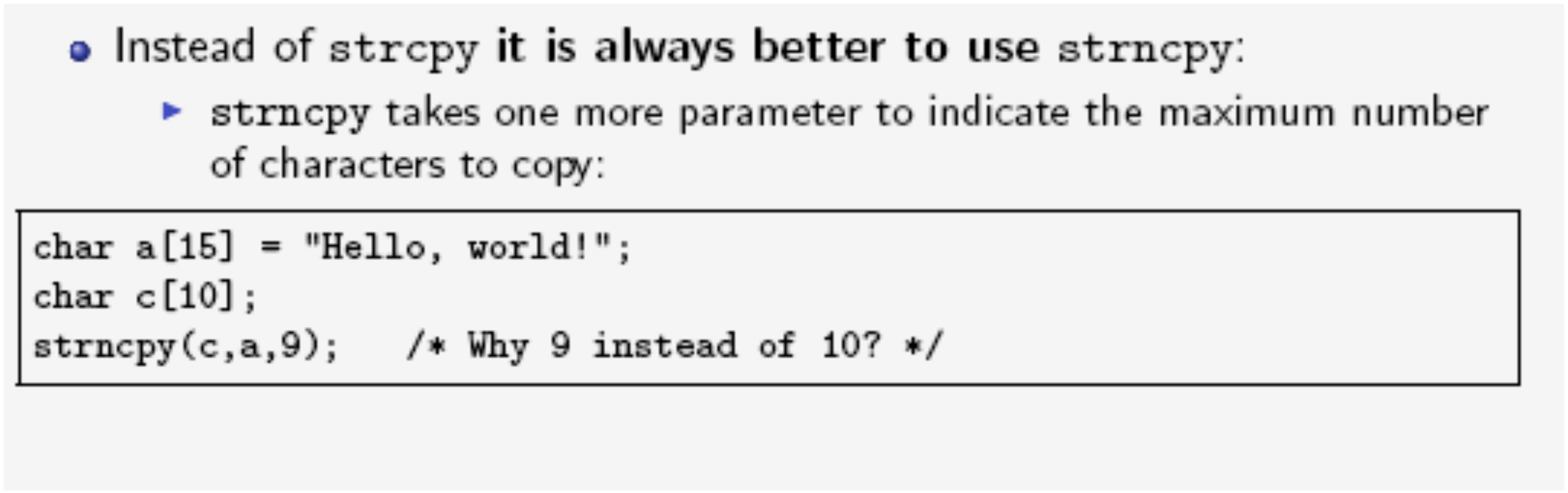
- There are standard function to manipulate strings:
  - `strcpy(destination, source)` will copy string **source** into string **destination**:

```
char a[15] = "Hello, world!\n";
char b[15];
strcpy(b,a);
```

☞ Attention: `strcpy` does **not** check that **destination** is large enough to accomodate **source**.

```
char c[10];
strcpy(c,a);  /* This will get you in BIG trouble */
```

Manipulating Strings *(cont.)*



- Instead of `strcpy` **it is always better to use** `strncpy`:
  - `strncpy` takes one more parameter to indicate the maximum number of characters to copy:

```
char a[15] = "Hello, world!";
char c[10];
strncpy(c,a,9);    /* Why 9 instead of 10? */
```

Comparison Operators

- The following operators are defined for basic data types:

```
if (a == b) { ... }
if (a != b) { ... }
if (a <  b) { ... }
if (a <= b) { ... }
if (a >  b) { ... }
if (a >= b) { ... }
if ((a==b) && (c>d)) {...}  /* logical AND */
if ((a==b) || (c>d)) {...}  /* logical OR */
```

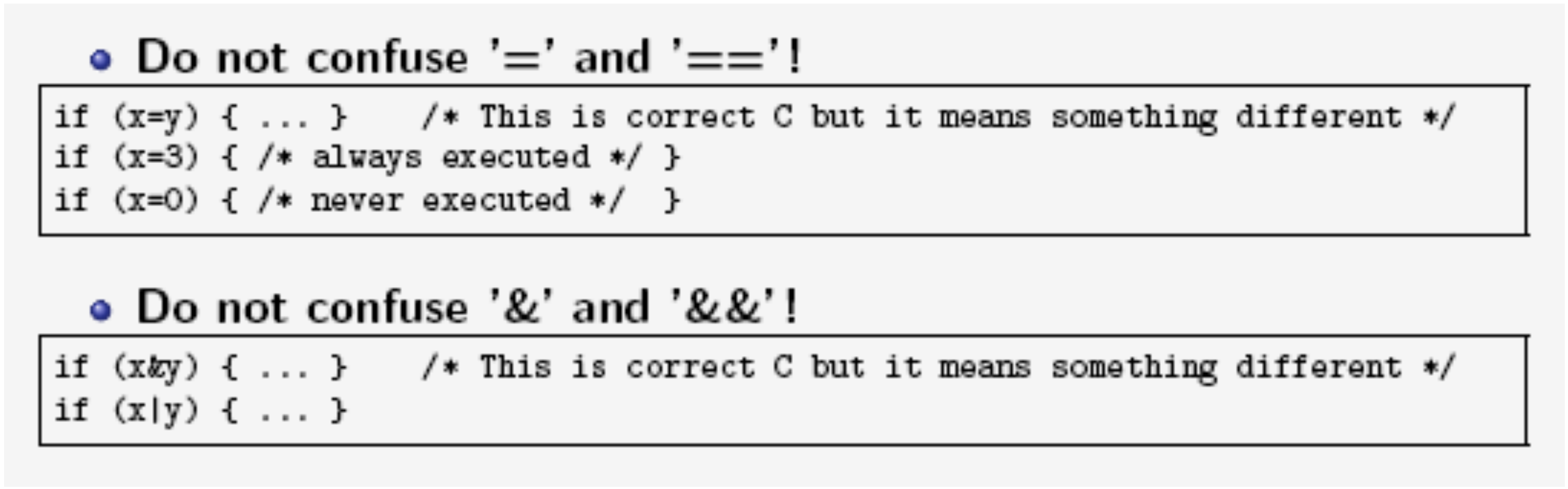
- There is no boolean type in C. We use integers instead:
  - 0 means FALSE
  - Any other value means TRUE

```
int x;
if (x)  {...}           /* Equivalent to: if (x!=0) {...} */
if (!x) {...}           /* Equivalent to: if (x==0) {...} */
```

Example

```
#include <stdio.h>

main()
{
	int x = 5;
	int y = 3;

	if (x=y){
		printf("x is equal to y, x=%d, y=%d\n", x, y);
	}
	else{
		printf("x is not equal to y, x=%d, y=%d\n", x, y);
	}

} // Then try again by replacing (x=y) with (x == y) in the if statement
```

Classical Bugs



- Do not confuse '=' and '=='!

```
if (x=y) { ... }    /* This is correct C but it means something different */
if (x=3) { /* always executed */ }
if (x=0) { /* never executed */  }
```

- Do not confuse '&' and '&&'!

```
if (x&y) { ... }    /* This is correct C but it means something different */
if (x|y) { ... }
```

Exercise:

- (7 & 8) vs (7 && 8)
- (7 | 8) vs (7 || 8)

Loops

```
while (x>0){
...
}

do{
...
} while (x>0);

for (x=0; X<3;x++) {...}
```

Functions

- In C, functions can be defined in two ways:

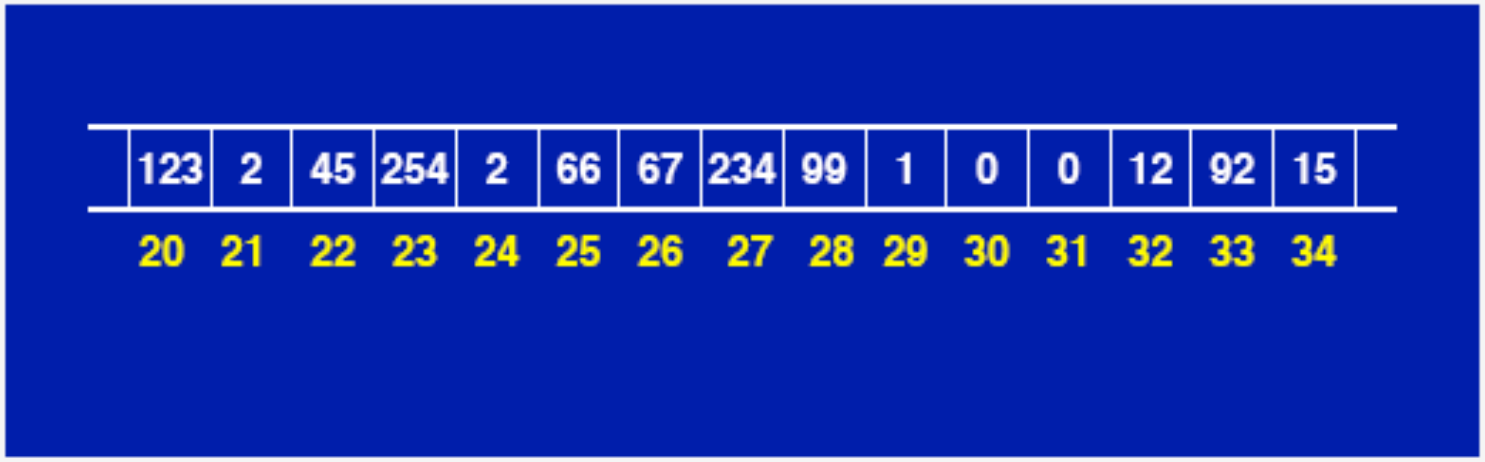
```
int foo() {                        /* function foo returns an int */
  ...
  return 123;
}

void bar(int p1, double p2) {     /* function bar returns nothing */
  ...
}
```

- Calling a function is easy:

```
int i = foo();  /* call function foo() */
bar(2, -4.321); /* call function bar() */
```

Memory Manipulation in C

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

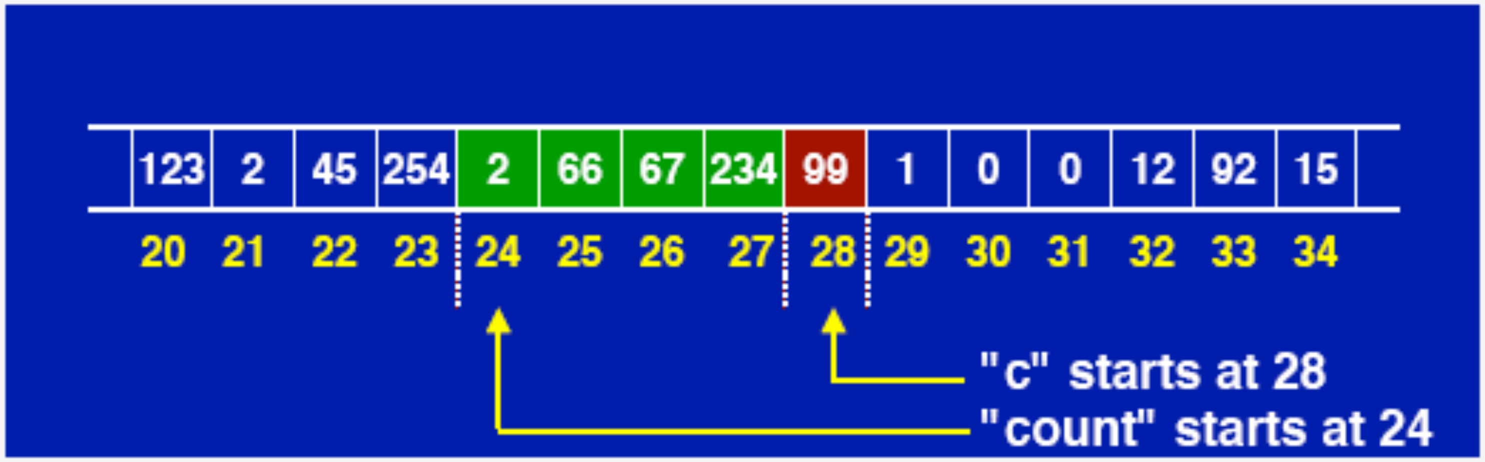


Memory Manipulation in C

- When you define variables:

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

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



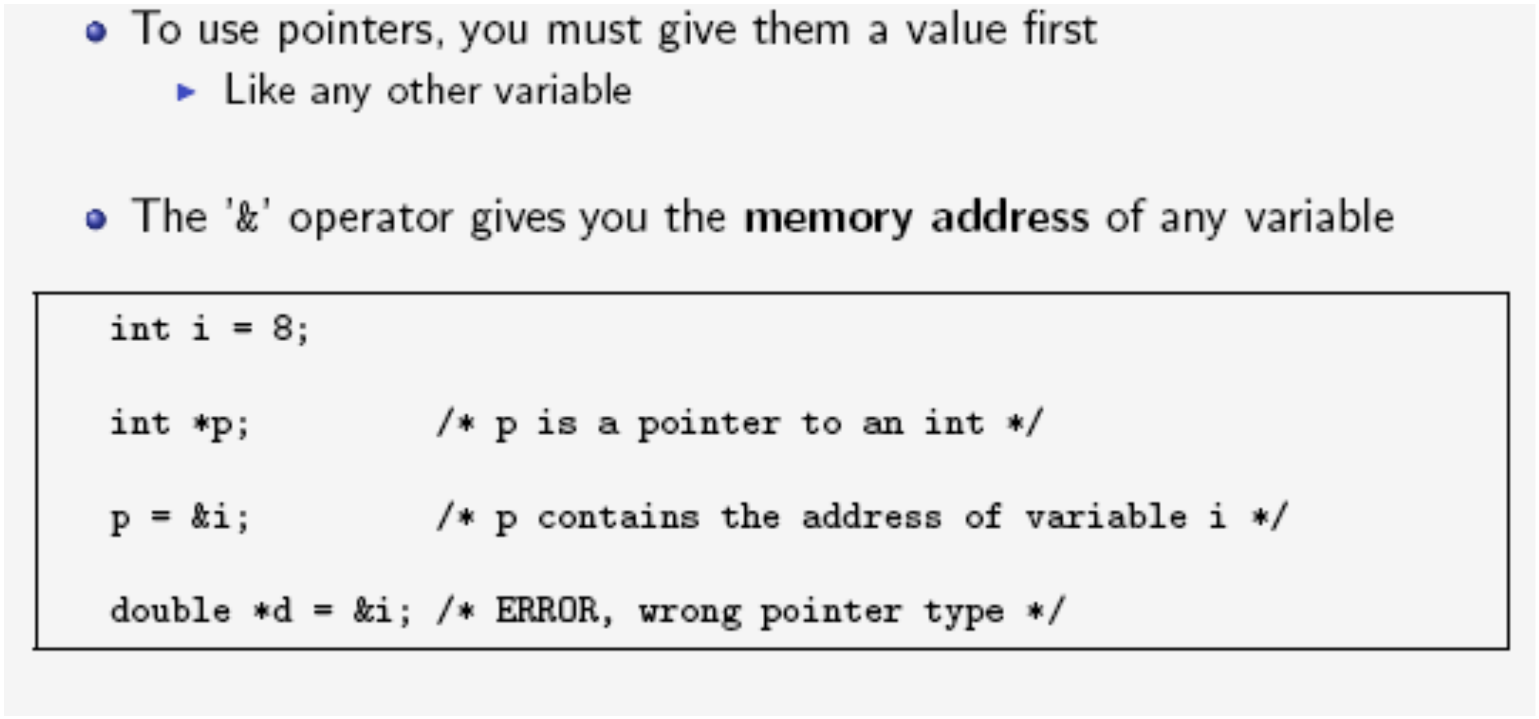
Memory Manipulation in C

- As a result, each variable has two properties:
  1. The '**value**' stored in the variable
     - ☞ If you use the name of the variable, you refer to the variable's value
  2. The '**address**' of the memory used to store this value
     - ★ Similar to a reference in Java (but not exactly the same)
     - ☞ A variable that stores the address of another variable is called a **pointer**

- Pointers can be declared using the * character

```
int *ptr;                  /* Pointer to an int */
unsigned char *ch;         /* Pointer to an unsigned char */
struct ComplexNumber *c;   /* Pointer to a  struct ComplexNumber */
int **pp;                  /* Pointer to a  pointer to an int */
void *v;                   /* Pointer to anything (use with care!) */
```

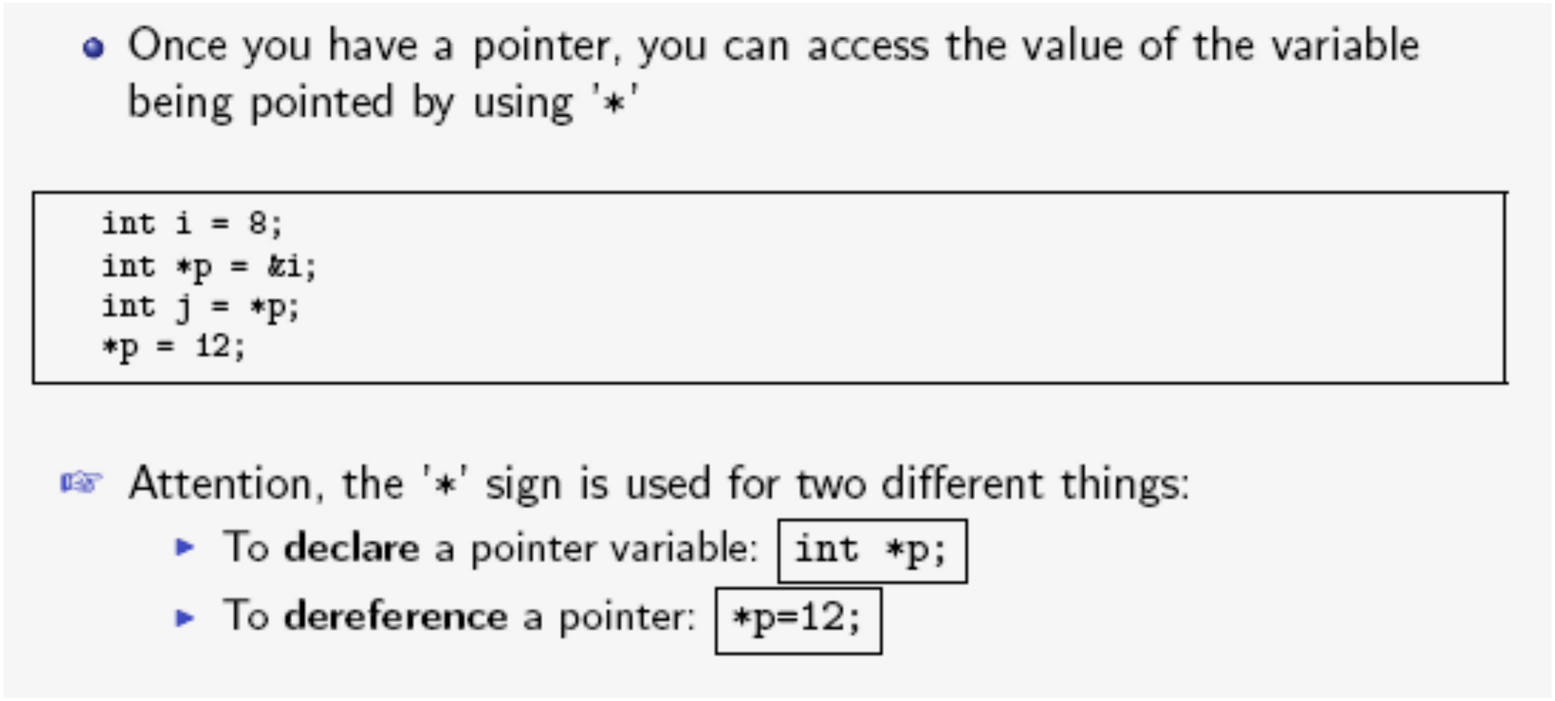
Defining Pointers



- To use pointers, you must give them a value first
  - Like any other variable
- The '&' operator gives you the **memory address** of any variable

```
int i = 8;

int *p;          /* p is a pointer to an int */

p = &i;          /* p contains the address of variable i */

double *d = &i; /* ERROR, wrong pointer type */
```

Using Pointers



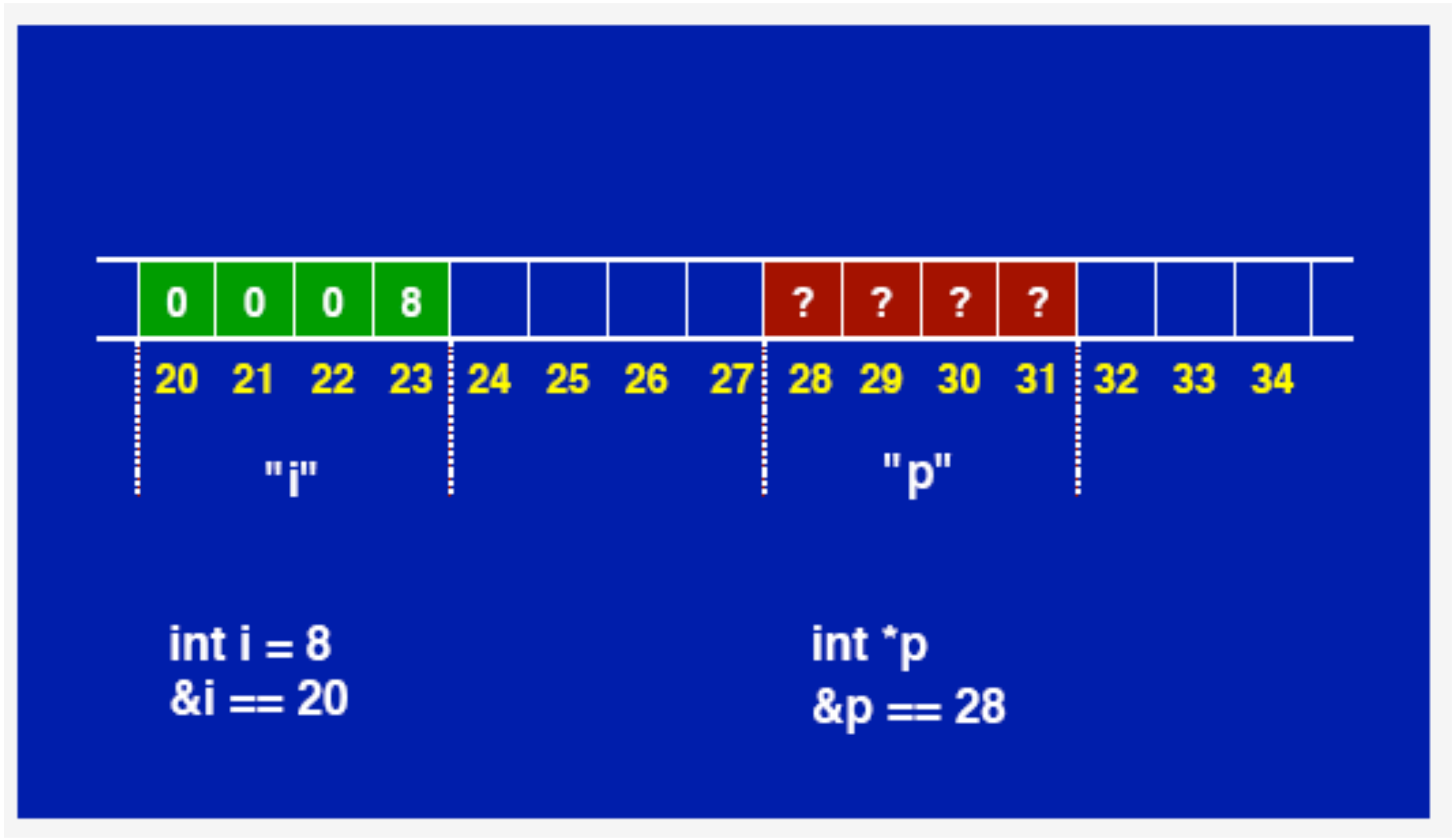
- Once you have a pointer, you can access the value of the variable being pointed by using '*'

```
int i = 8;
int *p = &i;
int j = *p;
*p = 12;
```

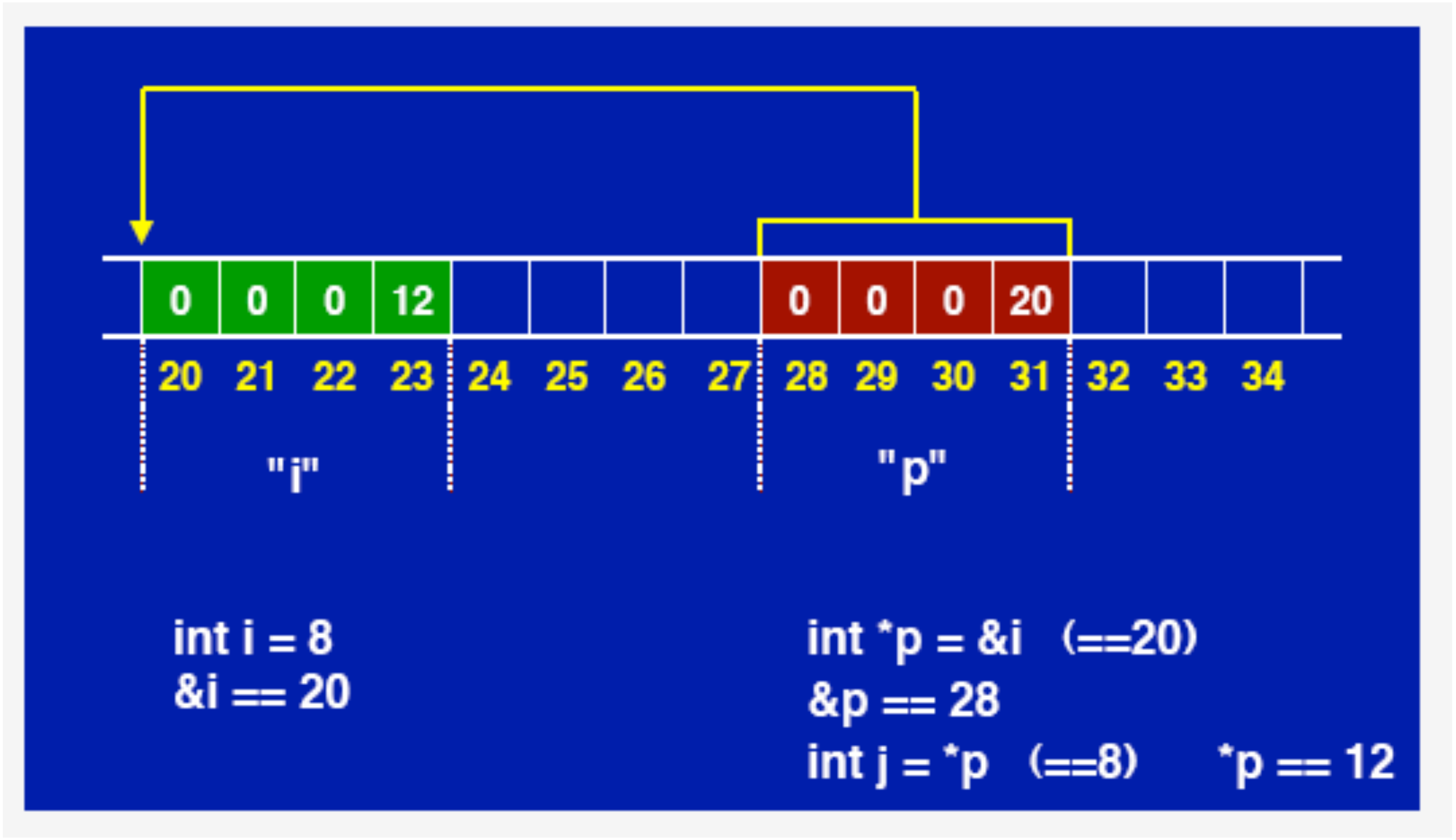
☞ Attention, the '*' sign is used for two different things:

- To **declare** a pointer variable: `int *p;`
- To **dereference** a pointer: `*p=12;`

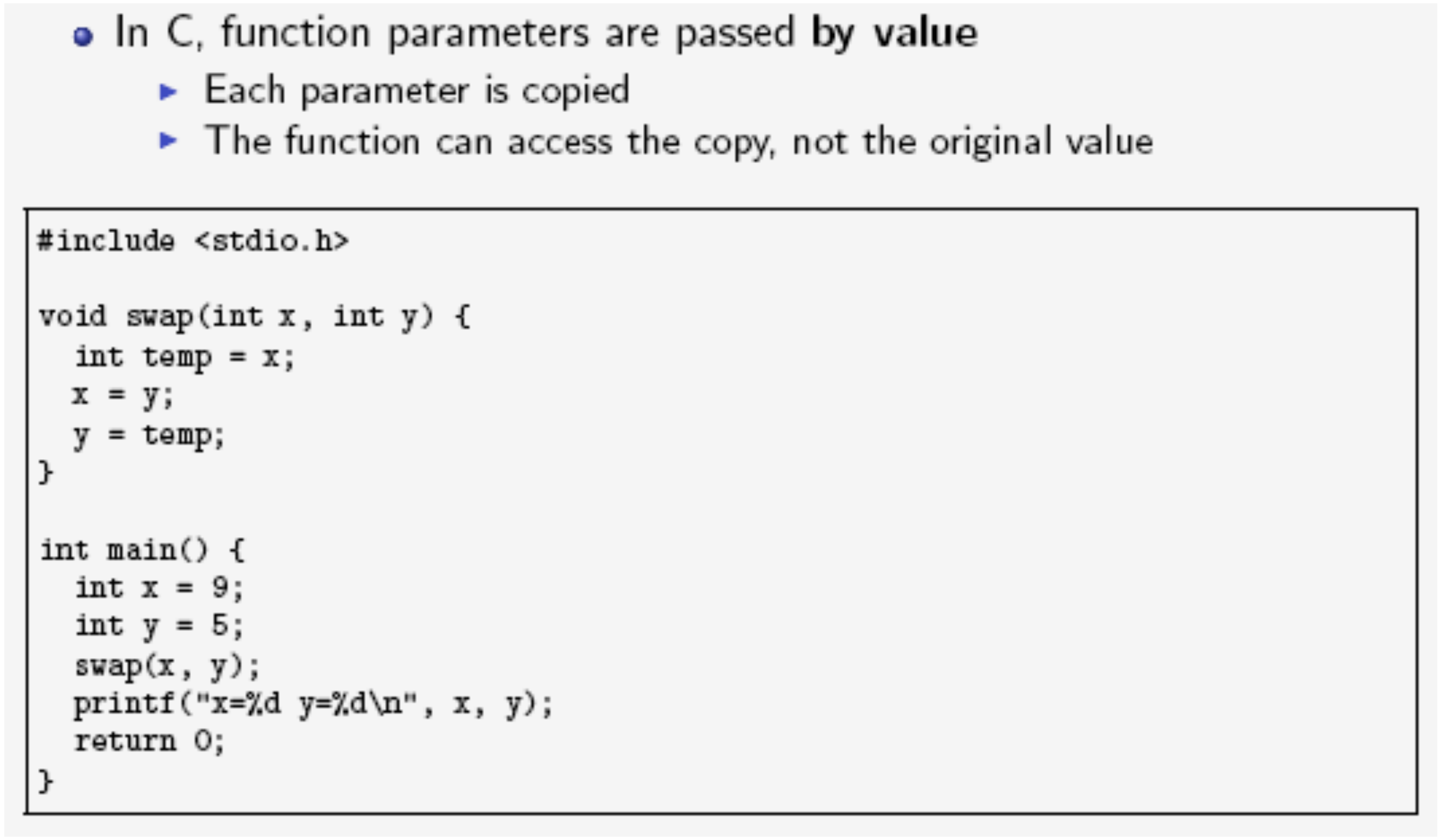
Using Pointers



Using Pointers



Parameter Passing in C



- In C, function parameters are passed **by value**
  - Each parameter is copied
  - The function can access the copy, not the original value

```
#include <stdio.h>

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

int main() {
  int x = 9;
  int y = 5;
  swap(x, y);
  printf("x=%d y=%d\n", x, y);
  return 0;
}
```

Parameter Passing in C

- In C, function parameters are passed **by value**
  - Each parameter is copied
  - The function can access the copy, not the original value

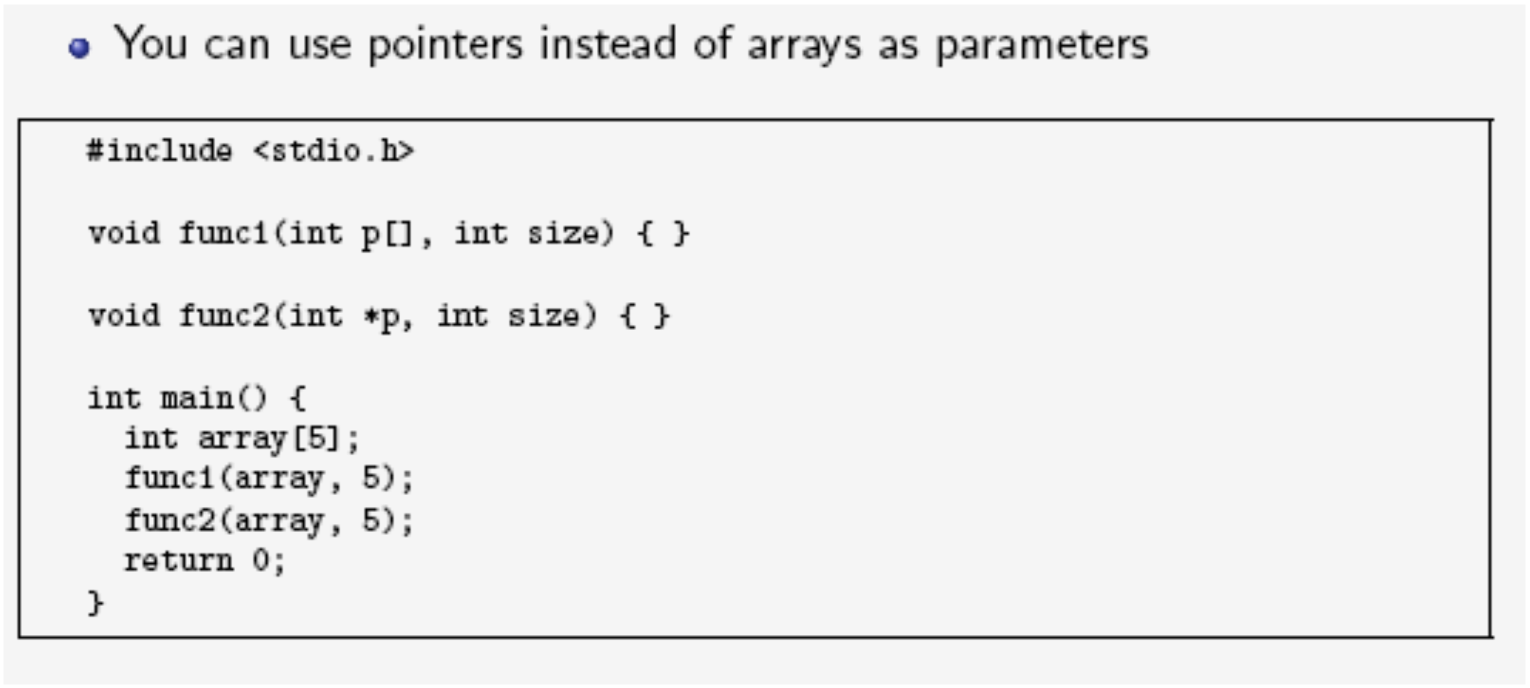
```
#include <stdio.h>

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

int main() {
  int x = 9;
  int y = 5;
  swap(x, y);
  printf("x=%d y=%d\n", x, y); /* This will print: x=9 y=5 */
  return 0;
}
```

Parameter Passing in C

- To pass parameters by reference, use pointers
  - The pointer is copied
  - But the copy still points to the same memory address

```
#include <stdio.h>

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

int main() {
  int x = 9;
  int y = 5;
  swap(&x, &y);
  printf("x=%d y=%d\n", x, y); /* This will print: x=5 y=9 */
  return 0;
}
```

Arrays and Pointers



- 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() {
  int array[5];
  func1(array, 5);
  func2(array, 5);
  return 0;
}
```

Arrays and Pointers

- You can even use array-like indexing on pointers!

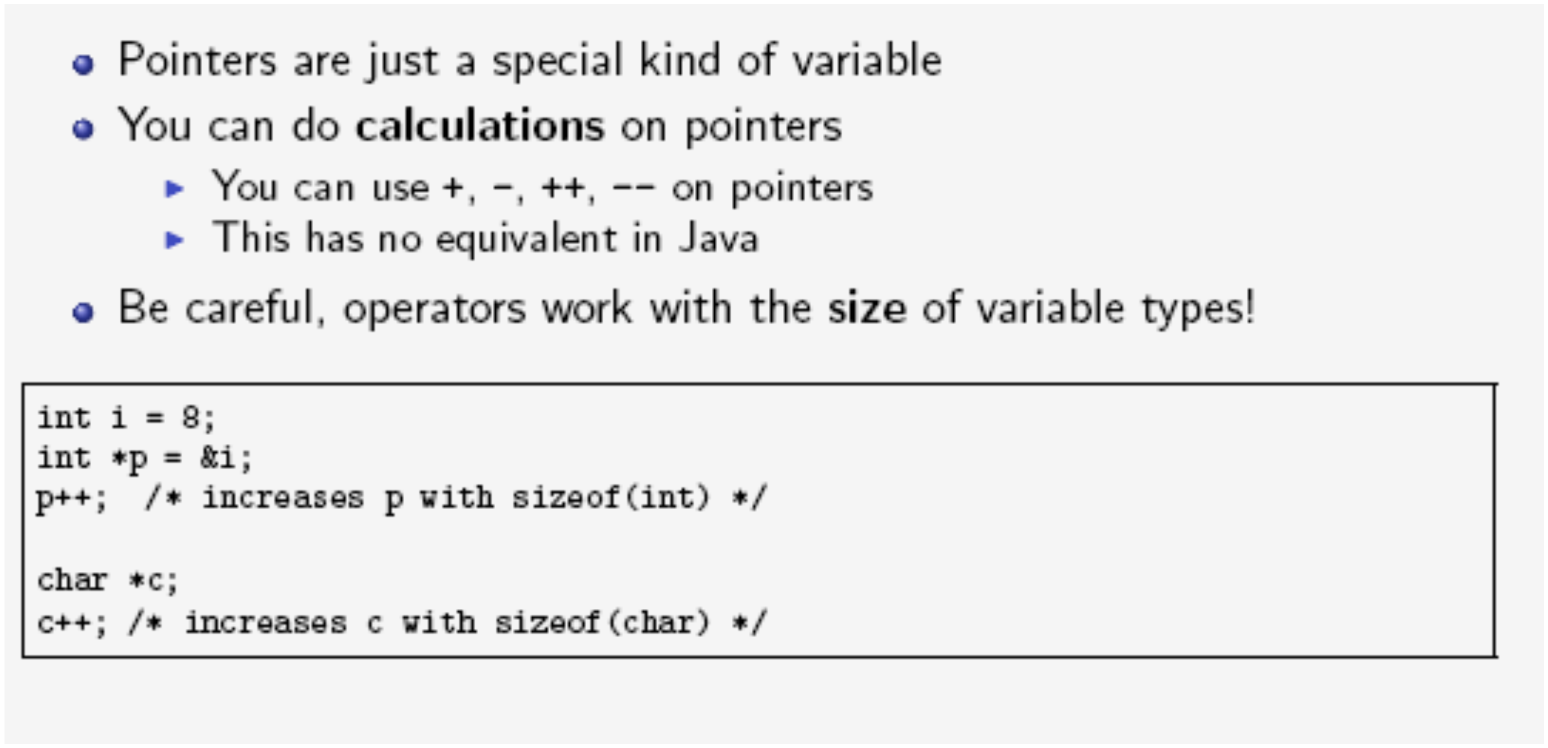
```
void clear(int *p, int size) {
    int i;
    for (i=0;i<size;i++) {
         p[i] = 0;
    }
}

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

Arrays and Pointers

- So a string is in fact just a pointer to a character array:

```
int main() {
   char s1[32] = "Hello, world!\n";
   char *s2;
   char s3[32];
   s2 = s1;              /* s1 and s2 point to the same character array */
   strncpy(s3,s1,31); /* s3 contains a copy of s1 */
```

Pointer Arithmetic



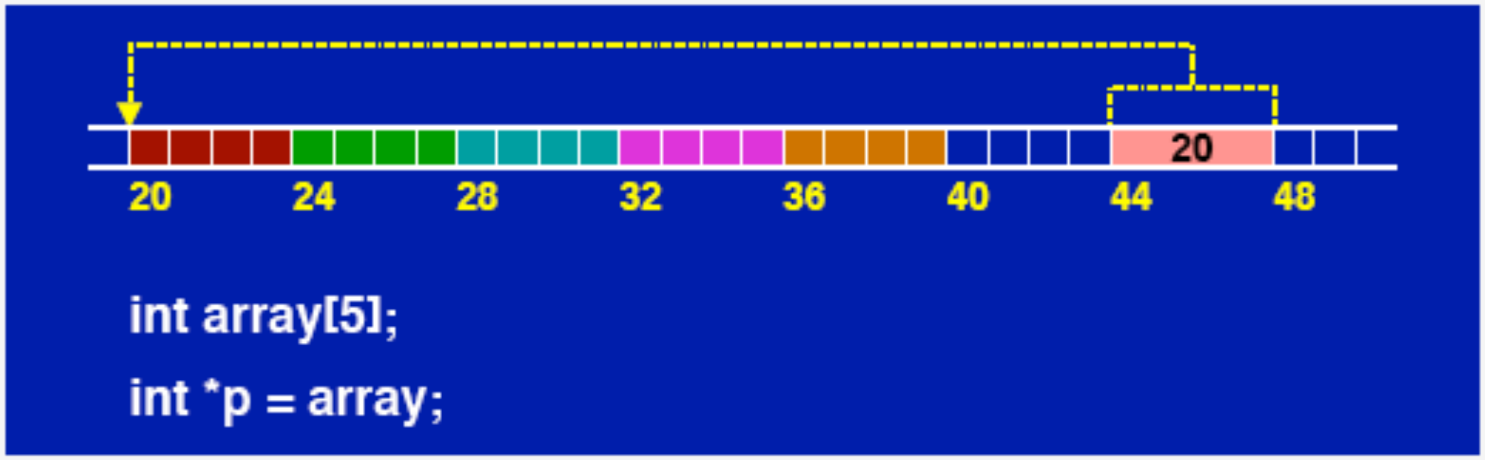
- Pointers are just a special kind of variable
- You can do **calculations** on pointers
  - You can use +, −, ++, −− on pointers
  - This has no equivalent in Java
- Be careful, operators work with the **size** of variable types!

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

char *c;
c++; /* increases c with sizeof(char) */
```

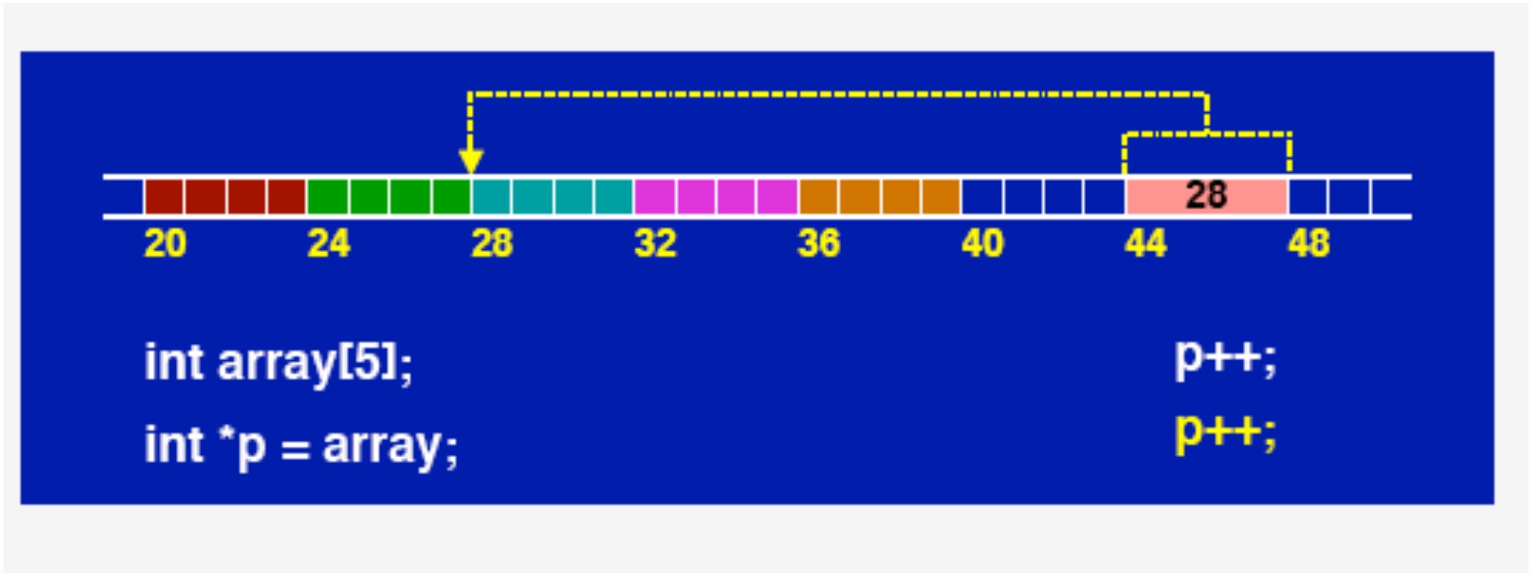
Pointer Arithmetic

- 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++;
}
```

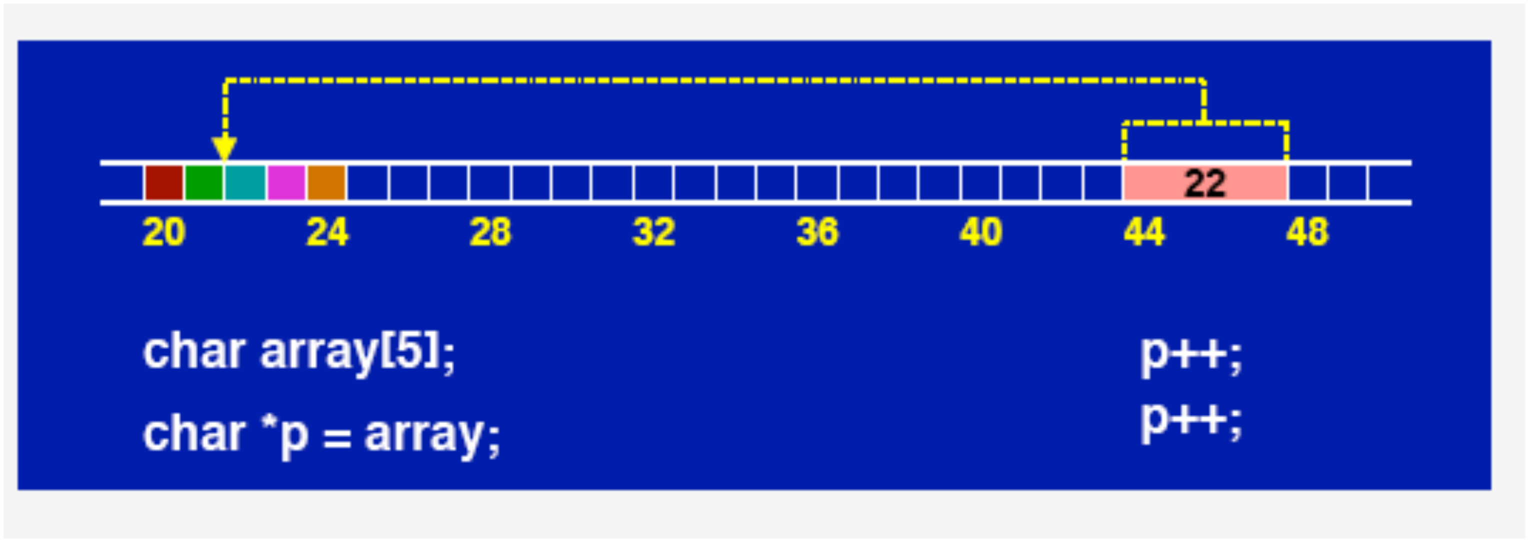


Pointer Arithmetic



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

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



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

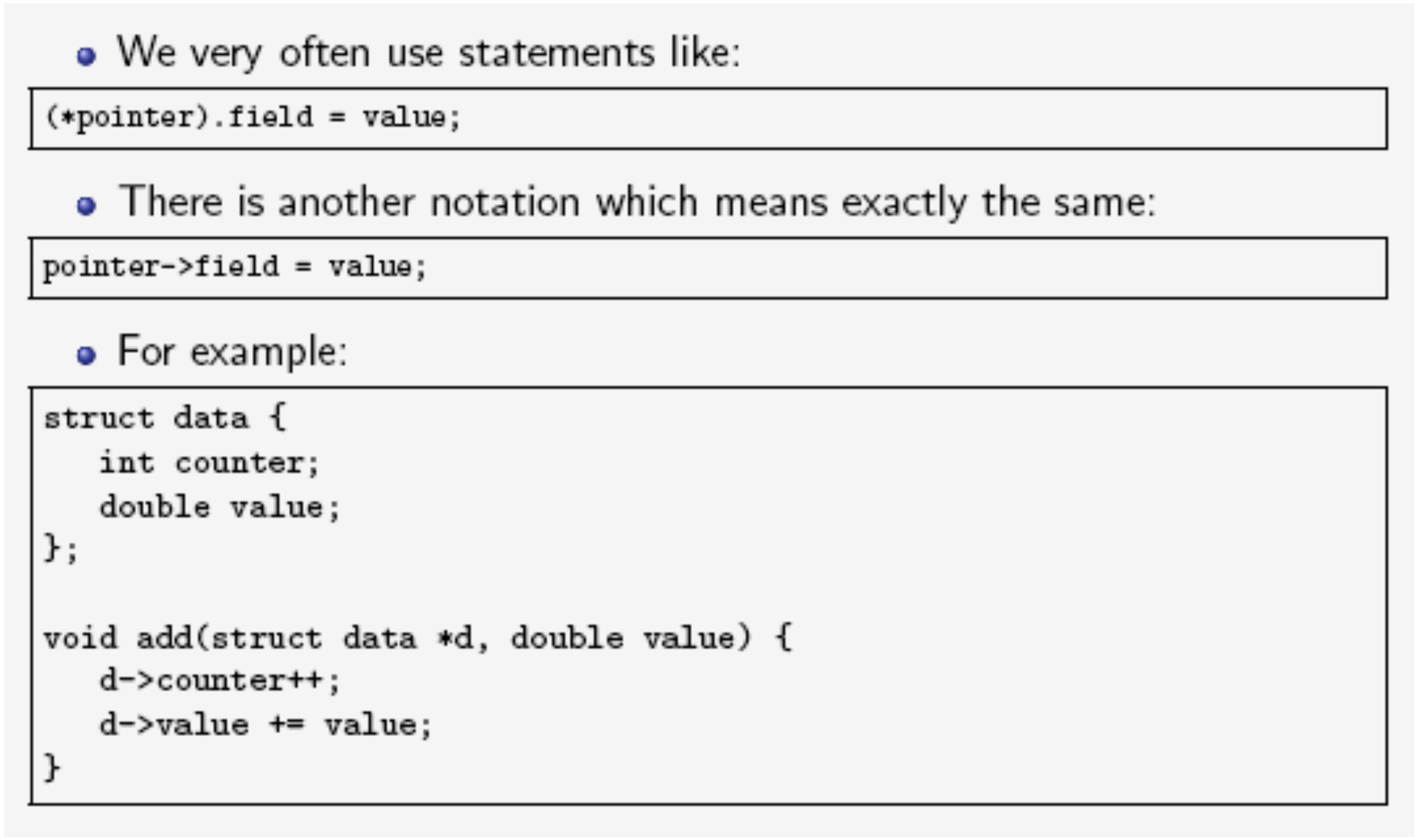
```
p++;
p++;
```

Structures

- You can build higher-level data types by creating structures:

```
struct Complex {
  float real;
  float imag;
};
struct Complex number;
number.real = 3.2;
number.imag = -2;

struct Parameter {
  struct Complex number;
  char description[32];
};
struct Parameter p;
p.number.real = 42;
p.number.imag = 12.3;
strncpy(p.description, "My nice number", 31);
```

Pointers to Structures



- We very often use statements like:

```
(*pointer).field = value;
```

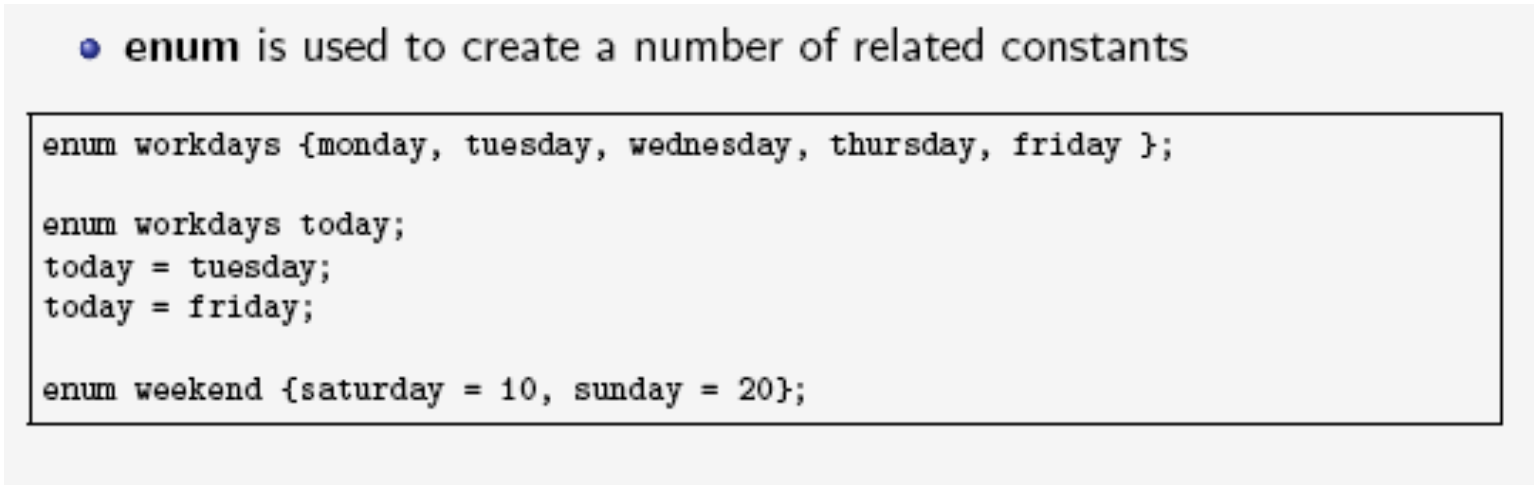
- There is another notation which means exactly the same:

```
pointer->field = value;
```

- For example:

```
struct data {
   int counter;
   double value;
};

void add(struct data *d, double value) {
   d->counter++;
   d->value += value;
}
```

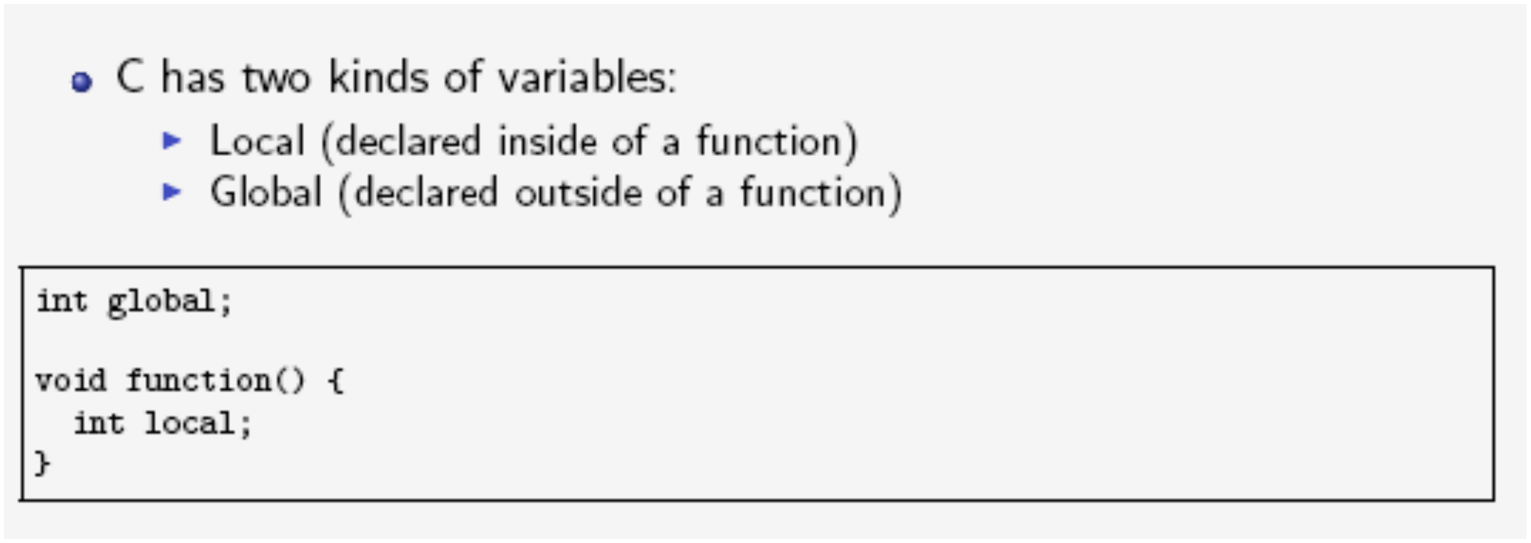
Enumerations



- **enum** is used to create a number of related constants

```
enum workdays {monday, tuesday, wednesday, thursday, friday };

enum workdays today;
today = tuesday;
today = friday;

enum weekend {saturday = 10, sunday = 20};
```

Variables



- C has two kinds of variables:
  - Local (declared inside of a function)
  - Global (declared outside of a function)

```
int global;

void function() {
  int local;
}
```

Static Local Variables

- Declaring a static variable means it will persist across multiple calls to the function

```
void foo() {
  static int i=0;
  i++;
  printf("i=%d\n",i);  /* This prints the value of i on the screen */
}

int main() {
  int i;
  for (i=0;i<3;i++) foo();
}
```

This program will output this:

```
i=1
i=2
i=3
```

Non-static Local Variables

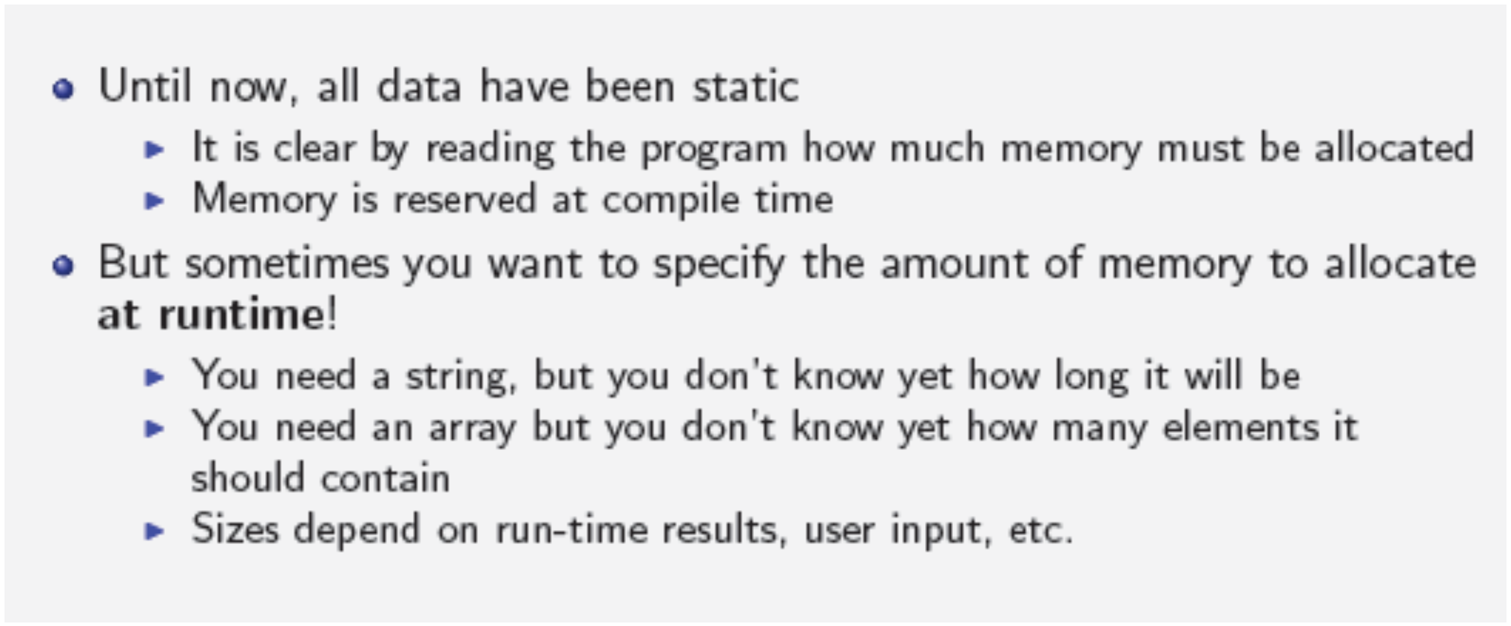
- If ***i*** is not static, the same example program (from prev. slide) will output:
  - i=1
  - i=1
  - i=1

Global Variables

Global variables have file scope:

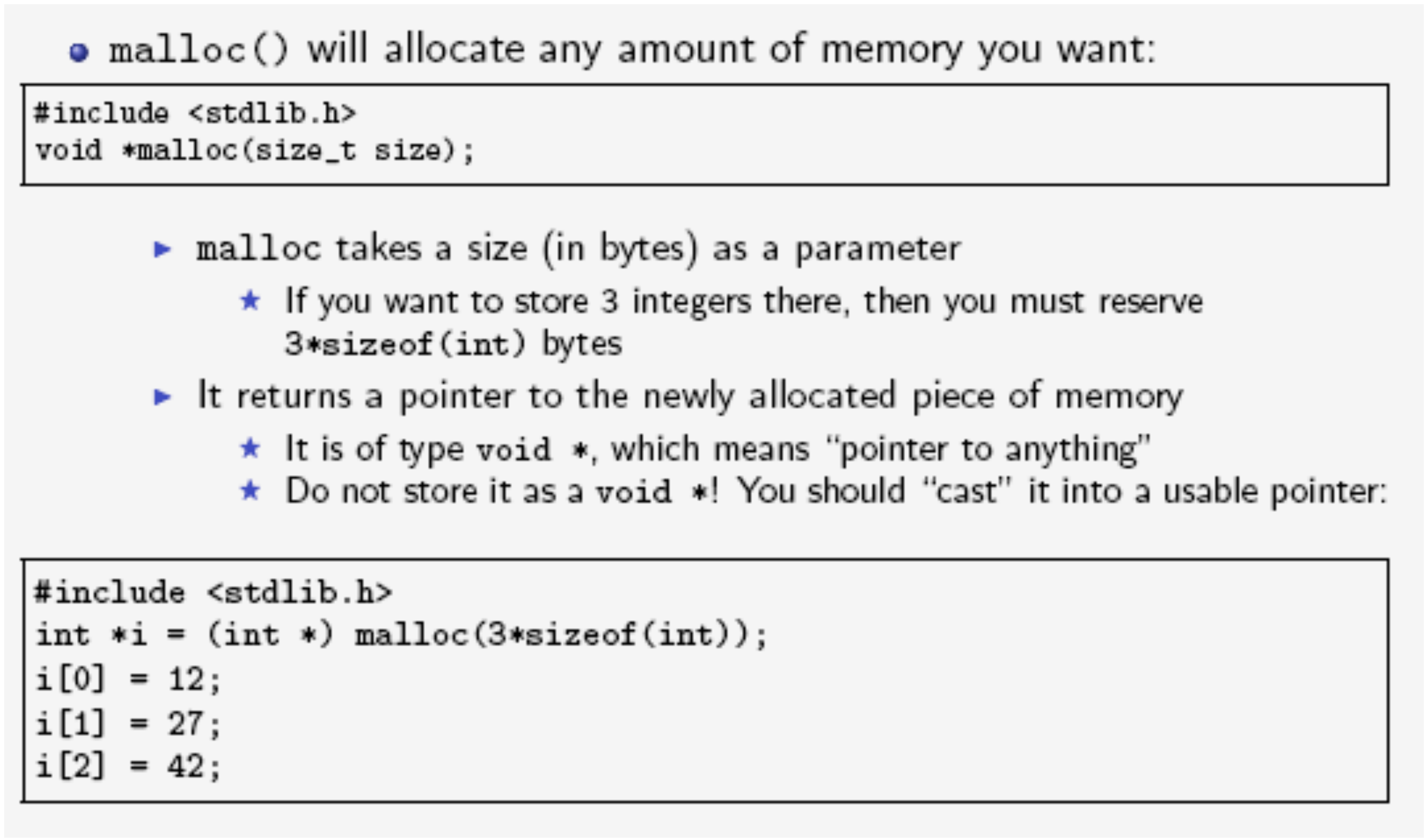
```
int i=0;

void foo() {
    i++;
    printf("i=%d\n",i);
}

int main() {
    for (i=0;i<3;i++) foo();
}
```

Dynamic Memory Management



- Until now, all data have been static
  - It is clear by reading the program how much memory must be allocated
  - Memory is reserved at compile time
- But sometimes you want to specify the amount of memory to allocate **at runtime**!
  - You need a string, but you don't know yet how long it will be
  - You need an array but you don't know yet how many elements it should contain
  - Sizes depend on run-time results, user input, etc.

Dynamic Memory Management



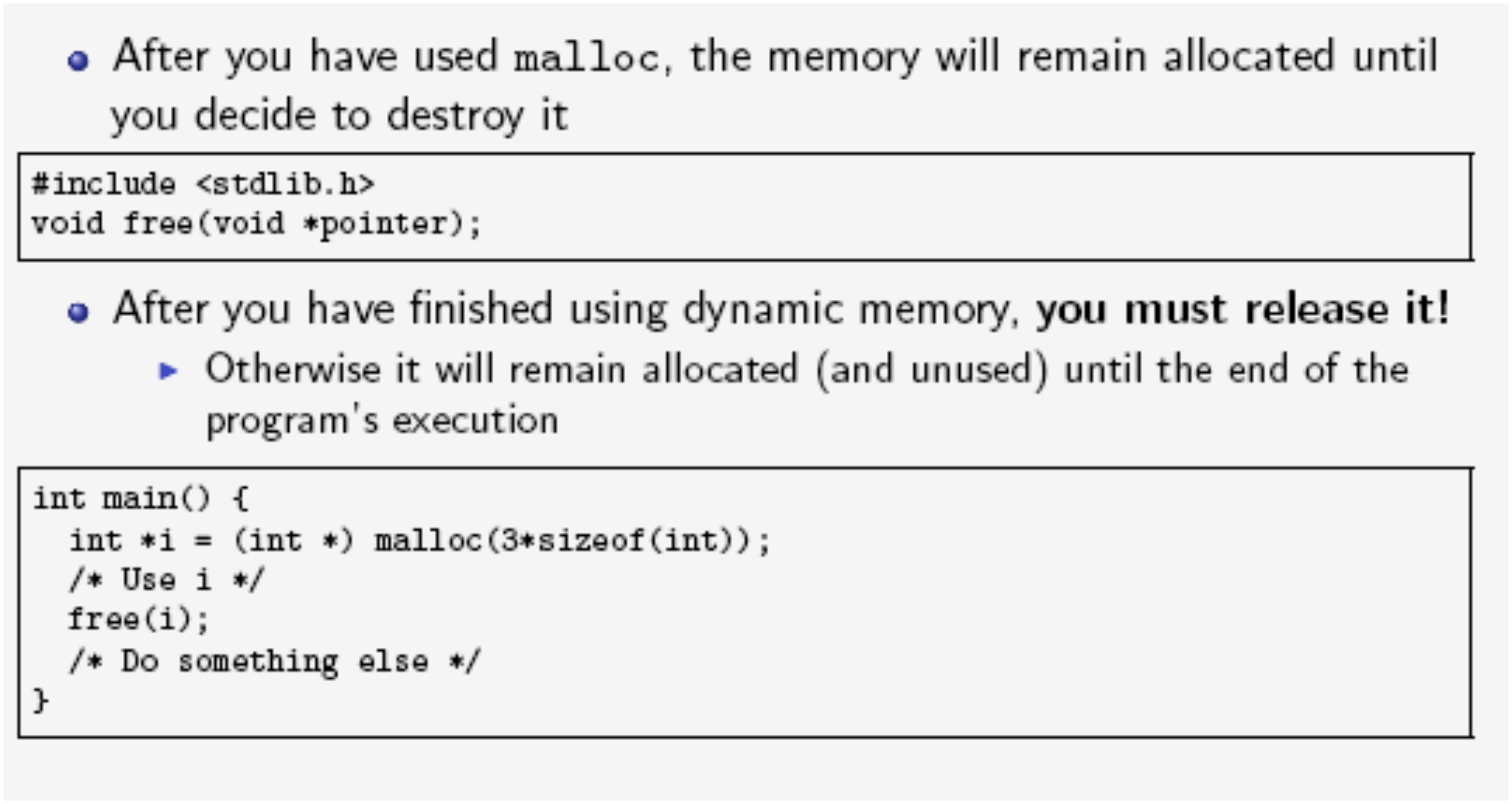
- `malloc()` will allocate any amount of memory you want:

```
#include <stdlib.h>
void *malloc(size_t size);
```

  - `malloc` takes a size (in bytes) as a parameter
    - If you want to store 3 integers there, then you must reserve `3*sizeof(int)` bytes
  - It returns a pointer to the newly allocated piece of memory
    - It is of type `void *`, which means "pointer to anything"
    - Do not store it as a `void *`! You should "cast" it into a usable pointer:

```
#include <stdlib.h>
int *i = (int *) malloc(3*sizeof(int));
i[0] = 12;
i[1] = 27;
i[2] = 42;
```

Dynamic Memory Management



- After you have used `malloc`, the memory will remain allocated until you decide to destroy it

```
#include <stdlib.h>
void free(void *pointer);
```

- After you have finished using dynamic memory, **you must release it!**
  - Otherwise it will remain allocated (and unused) until the end of the program's execution

```
int main() {
  int *i = (int *) malloc(3*sizeof(int));
  /* Use i */
  free(i);
  /* Do something else */
}
```

Dynamic Memory Management

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

```
int *createIntArrayWrong() {
  char tmp[32];
  return tmp;                          /* WRONG! */
}

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

int main() {
    int *array = createIntArray(10);
    /* ... */
    free(array);
    return 0;
}
```

Memory Leaks

- 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!

```
int main() {
  int *i = (int *) malloc(3*sizeof(int));
  i = 0;      /* Wooops, I lost the pointer to my dynamic memory */
  free(???);  /* It is too late to free my dynamic memory */
```

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

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

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

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

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

malloc Example

```
int main ()
{
    int x = 11;
    int *p, *q;

    p = (int *) malloc(sizeof (int));
   *p = 66;
    q = p;
    printf ("%d %d %d\n", x, *p, *q);
    x = 77;
   *q = x + 11;
    printf ("%d %d %d\n", x, *p, *q);
    p = (int *) malloc(sizeof (int));
   *p = 99;
    printf ("%d %d %d\n", x, *p, *q);
}
```

```
$./malloc
11 66 66
77 88 88
77 99 88
```

free Example

```
int main ()
{
   int x = 11;
   int *p, *q;
   p = (int *) malloc(sizeof (int));
  *p = 66;
   q = (int *) malloc(sizeof (int));
  *q = *p - 11;
   free(p);
   printf ("%d %d %d\n", x, *p, *q);
   x = 77;
   p = q;
   q = (int *) malloc(sizeof (int));
  *q = x + 11;
   printf ("%d %d %d\n", x, *p, *q);
 p = &x;
   p = (int *) malloc(sizeof (int));
  *p = 99;
   printf ("%d %d %d\n", x, *p, *q);
 q = p;
   free(q);
   printf ("%d %d %d\n", x, *p, *q);
}
```

```
./free
11 ? 55
77 55 88
77 99 88
77 ? ?
```

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), and B. Knicki (WPI).