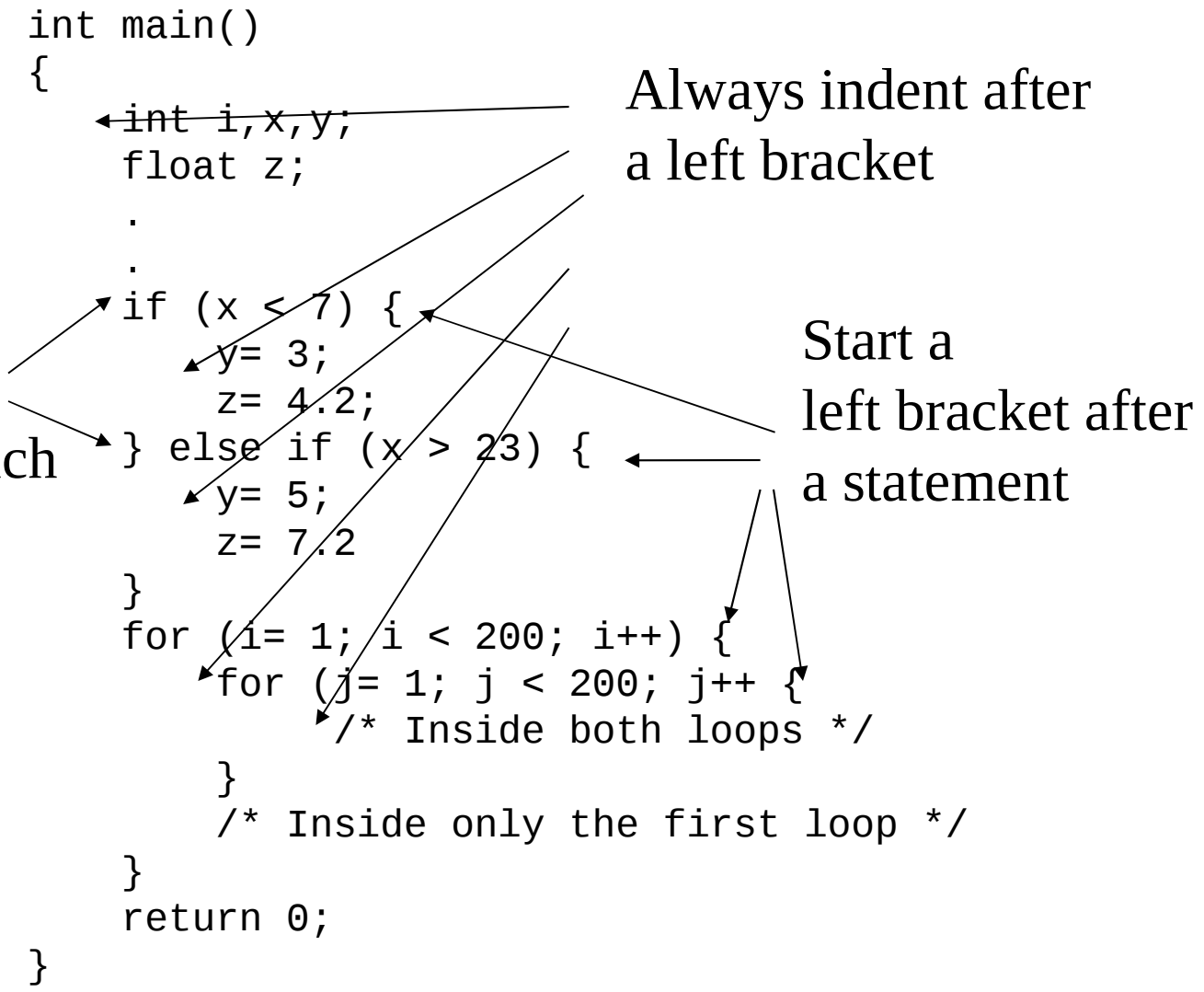
C Programming lecture 2

- Beautiful programs
- Greek algorithms
- Arrays of numbers
- Strings of characters
- Pointers (the really difficult bit of C)

Beauty is truth and truth beauty

- A program which is neatly formatted is easier to understand
- If you're staring in confusion at a load of {s and }s and don't know whether you need another } your program is probably UGLY
- Indenting consistently is especially important. (in functions, for, while if and else).
- One of our most important goals as a programmer is to write something that other people can read.

Layout of a program (brace style)



NOTE: Not all code on these overheads follows this style for space reasons

MAGIC numbers in programs

- A lot of programs contain what programmers sometimes call MAGIC nos

```
g= 43.2 * a + 7.1;
for (i= 7; i < 103; i+=2) {
    printf ("%d\n",i*7);
}
```

This makes code look UGLY and it is confusing to the reader. It is better to give some idea of what these numbers mean in the program.

Enum

- We can use the enum command to define int s and chars. It looks like this:

```
enum {
    MAX_LEN= 100,
    LETTERX= 'x',
    NEWLINE= '\n'
};
```

By convention we use all capitals for enum constants. That way we can tell them from variables. We can now use the enum constant wherever we could use an int or char

```
for (i= 0; i < MAX_LEN; i++)
    printf (“%c”,LETTERX);
```

#define

- This preprocessor command replaces one thing with another before compilation

NOTE – NO semi colon here!

```
#define PI 3.14
#define GRAV_CONST 9.807
#define HELLO_WORLD "Hello World!\n"
```

Now, anywhere in the program we use PI or GRAV_CONST it will be replaced with the replacement string BEFORE the real compiler starts (that's why we call it pre-processing)

```
c= 2.0 * PI * r;
a= GRAV_CONST * (m1*m2)/ (r * r);
printf (HELLO_WORLD);
```

Const

- Another solution is to use the `const` keyword.
- This is perhaps the best solution but it sometimes causes problems on older compilers.

```
/* Approximate value of PI */
const double PI=3.14;
/* Maximum iterations to be performed
before exiting */
const int MAX_ITER=1000;
```

The ARRAY

- An array in C is a group of similar variables. For example 200 ints or 45 chars
- Arrays use square brackets like so:

```
int some_nums[200];
char bunch_o_chars[45];

some_numbers[3]= 5;
printf ("Element 3 is %d\n",some_nums[3]);
bunch_o_chars[0]= 'a';
```

Unfortunately, in C, we must give the length when we declare the array

ARRAYS CAN BE MODIFIED BY FUNCTIONS!
For the reason see later in the lecture

Common Error

- If we declare an array in C to be 100 elements the array is numbered from 0 to 99. EVERYBODY at some point makes this mistake:

```
int a[100]; /*Declare 100 ints */
a[0]= 52;
a[1]= 3;
a[100]= 5;  /* This is a common error
               There is NO element 100
               but your program will compile
               and run*/
```

Passing arrays to functions

- We prototype a function which accepts an array like this:

```
void process_array (int []);
int calc_array (char[]);
```

- And write the function like this:

```
void process_array (int all_nums[])
{
    all_nums[1]= 3;
    .
    .
}
```

- And call the function like this:

```
int some_numbers [100];
process_array(some_numbers);
```

- Note that we CAN’T return an array from a function (but see later).

Strings in C

- When we want to store and print text in C we use a *string*
- A string is an array of type char which has '\0' as the last character.
- '\0' is a special character (like '\n') which means "stop now".
- We can initialise a string using = but we can't set a string using this at other times (use sprintf)!
- We can print a string with %s:

```
char hi_there[]= "Hello World!\n";
printf ("%s", hi_there);
```

Useful functions with strings

- We can find a lot of useful functions in the library `string.h`

```
#include <string.h>
#include <stdio.h>
int main()
{
   char test[100];
   char test2[]= "World!\n";
   strcpy(test,"Hello");
   strcat(test,test2);
   if (strcmp(test,"dave") == 0)
       printf ("Test is same as Dave\n");
   printf ("Length of test is %d\n",
     strlen (test));
}
```

Common Error

- Remember, our string must be big enough to HOLD what we put in it.

```
char small_string[]= "Hey there";
sprintf (small_string,"Hello World!\n");
```

This will almost certainly cause problems. The first statement sets up a string which is only big enough to hold 10 characters [Why 10? Think about it.] The second statement tries to put 14 characters into it. Disaster!

The scanf statement

- Scanf can be used like printf but to read instead of write.
- It is a confusing way to read data from the user (we will see why in later lectures)

```
int number, check;
check= scanf ("%d",&number);
if (check != 1) {
    printf ("Error!\n");
    return –1;
}
```

The scanf statement

- scanf uses format string to read data
- %d – int type, %f – float and %lf for double
- for printf it is %f for both float and double

Pass by reference/Pass by value

- Normally when we send a variable to a function we make a COPY of the variable.
- This is called *pass by value*. A value is passed and this copy of the variable arrives at the function.
- Sometimes, like in scanf we want to change the variable inside the function.
- In this case we need to do something different: *pass by reference*.
- This is what the & character is for.

```
void incrementInt( int &a){
        a=a+1;
}
int b=5;
incrementInt(b);
```

What is the value of b after this function call?

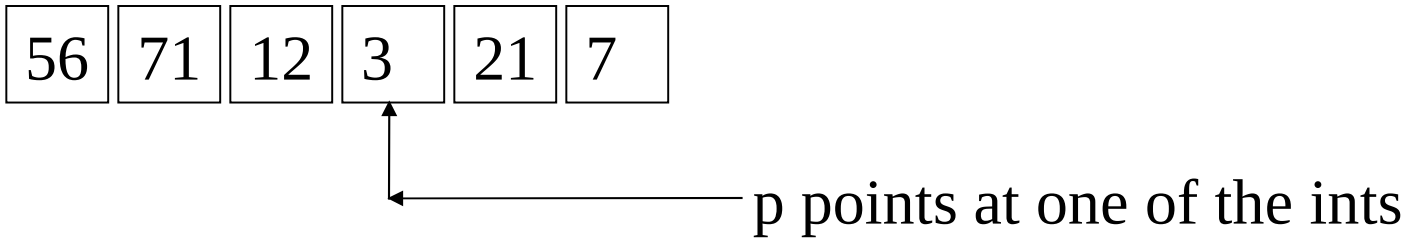
What are pointers?

- Pointers are one of the most difficult topics to understand in C.
- Pointers "point at" areas of your computer's memory.

```
int *p;
```

says p is a pointer to an int

- Imagine your computer's memory as a series of boxes which all hold ints



& means "address of" or "point at me"

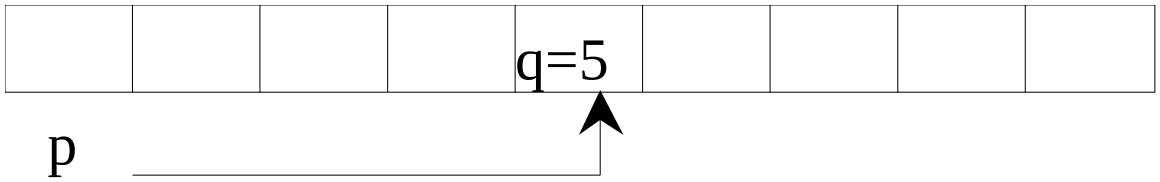
```
int *p;
int q= 5;
p= &q;
```

p is a pointer to an int

q is an int

p now points at q

We use `*p` to mean "the value p is pointing at"



* means value or "what am I pointing at?"

- Recall that we declare a pointer with a `*` use an `&` to get the "address of" (and convert a variable to a pointer)
- We use a `*` to get the value "pointed at"

```
int *p;
int q= 5;
p= &q;
*p= 6;
printf ("q is now %d\n",q);
printf ("p is now %d\n",*p);
```

What's the point of pointers?

- When we use & to pass a pointer to a variable into a function we CAN change the value of the variable within the function.
- This is called pass by reference.
- This is what was going on when we use & in `scanf`.
- We will also learn that arrays are nearly the same thing as pointers.

How we can change an argument within a function

```
void square_num (int *);

int main()
{
   int p= 5;
   square_num (&p);
   printf ("p is now %d\n",p);
   return 0;
}

void square_num (int *pnum)
{
    (*pnum)= (*pnum) * (*pnum);
}
```

Prototype a function taking a POINTER

Pass the address of p to our function

Now the function has changed p to 25

Remember * gives the value the pointer points at

Pointer Arithmetic 2

- A pointer is another type of array and we can mix between them in certain arrays
- If we define an array we can use pointers to access it

```
int i[7]; /* An array of 7 ints */
int *j;    /* A pointer to an int*/
j= i;     /* j points at the start of i*/
*j= 3;    /* Same as i[0]= 3 */
j= &i[0];  /* Same as j= i */
j= j+1;    /* Move j to point at i[1]*/
```

Pointer Arithmetic Test

```
int i[7];
int *j;
j= i;    /* J points at i[0] */
j= j+1;  /* Moves pointer */
*j= 3;
j= i;    /* Same as j= &i[0] */
*j= 10;
j[1]=4;
j= j+5;
j[1]= 5;
i= i+4;   /* This is an error */
j= j+7;   /* J points off end of array */
*j= 4;  /* This is an error */
```

An example of pointer

A small function to swap the value of two variables a and b.

```
/*this code will not work*/
void swap(int a , int b)
 {
     int temp;
     temp = a;
     a = b;
     b = temp;
 }
```

```
/*this code will work*/
void swap(int *a , int *b)
{
     int temp;
     temp = *a;
     *a = *b;
     *b = temp;
}
```

The left implementation passes the values of a and b the function, however the values of a and b will never be changed! The right implementation passes the address of a and b via pointers, which make it possible to change the values of a and b.

Relation between pointers and arrays

In C, is a very close connection between pointers and arrays. When you declare an array as:
int a[10];
You are in fact declaring a **constant** pointer a to the first element in the array. Therefore

- a==&a[0] and *a==a[0]
- Array indexing is equivalent to pointer arithmetic, that is a+i=&a[i] and *(a+i)==a[i]

Since a is a constant pointer, a++ or a- - will not work.

Pass an array to functions

An array can be passed to a function via a pointer and integer variable that indicates the size of the array.

```
int randdat(int *pa , int n){
  int i;
  for ( i=0 ; i< n ; ++i){
    *pa = rand()%n + 1;
    ++pa;
   }
}
```

In the above code,

```
*pa = rand()%n + 1;
++pa;
```

is equivalent to

```
*(pa+i)=rand()%n+1;
```

or

```
pa[i]=rand()%n+1;
```

The above function can also be declared as following:

```
int randdat(int pa[] , int n)
```

Integer operation on a pointer

Adding a integer j to a pointer p yields a pointer to the element that is j places after the one that p points to. So if p points to the array element a[i], then p + j points to a[i+j].