Lecture -3

- Arrays
- Strings
- Reading input from the user

Arrays

* In the last lecture, we introduced loop constructs, which let us perform the same actions repeatedly until a condition was met.
* It would be useful to be able to represent lists, or *arrays*, of values so that we can use loops to efficiently perform the same actions on each of them.
* C provides a syntax for creating *array variables* which do precisely this (their storage in memory is simply all the values in the array, one immediately after the other).

Declaring an Array

* An array variable is declared much like a scalar variable: the declaration starts with a type specifier, followed by the names we want to use for variables.
* To make a variable an array, we suffix the name with the number of elements we would like the array to hold, surrounded by [ ]
* We can mix array and scalar names in the same declaration, if their base value type is the same.

```
double a[5];
int b[4] = {0,4,2,27};

//q and z are just shorts
//h an array of 56 shorts
short q, h[56], z;
```

Declaring an Array

* We can give all the elements of an array initial values, provided in a comma-separated list surrounded by { }
* If we specify an initialiser list like this, then we can leave out the explicit size of the array between the [ ]s; the compiler will make an array exactly big enough to hold the list.

```
double a[5];
int b[4] = {0,4,2,27};

//q has size 2
double q[] = {0.1, 4.0e1};
```

Using Arrays

* Now we've made an array, how do we access the values in it?
* The values (*elements*) of an array are numbered, *starting at zero*.
* We specify an element using the same `[ ]` construct we used to declare our array.
* `a[4]` specifies the *5th* element of `a` (counting from 0).
* For maximum speed, C *does not check* that the array is long enough to contain the element you asked for!
* If `a` was of length 4, then our example would access memory outside the array, with undefined (and potentially bad) effects.

Array Example

```c
#include <stdio.h>

int main() {
    int dataArray[] = {5,4,3,2,1};
    int sum = 0 ;

    for (int i = 0; i < 5; i++)
        sum += dataArray[i];

    printf("Sum of the data is %d \n",sum);

    return 0;
    }
```

Array Example Continued

```c
#include <stdio.h>

int main() {
    int dataArray[] = {5,4,3,2,1};
    int sum = 0 ;

/*
    for (int i = 0; i < 5; i++)
      sum += dataArray[i];

Now lets replace for-loop with a do...while loop.
We will need to define a loop counter to end the looping */

//  int i = 5;
    int i = sizeof(dataArray) / sizeof(dataArray[0]);

    do {
     sum += dataArray[--i];
    } while (i > 0);

/*  while (i > 0)
        sum += dataArray[--i];
*/
    printf("Sum of the data is %d \n",sum);

    return 0;
    }
```

Comparing and Copying Arrays

- To compare two arrays - compare each element in turn using a loop. Note: `if(array1 == array2)` will always return 'false' because it is comparing the memory addresses of the two arrays which <u>are</u> different).

```c
#include <stdio.h>

int main() {
    int a[] = {5,4,3,2,1};
    int b[] = {0,0,0,0,0};
    int n = sizeof(a) / sizeof(a[0]);

    for (int i = 0; i < n; i++) {
     if(a[i] != b[i]) { // compare the two arrays
        puts("The arrays are different");
        break;
     }
    }
```

Multidimensional Arrays

- Arrays can also be created with more than one dimension, by writing a sequence of [] in the declaration.

Not:
twoDarray[3,3]
twoDarray[3;3]
etc

- `int twoDarray[3][3]; //3 by 3 array`
- Essentially, this creates an “array of arrays”, so
- `twoDarray[0]`
- Gives you the first row of the array (as an array of 3 ints)
- `twoDarray[0][0]`
- Gives you the first element of the array (by taking the first int in that first row)
- `twoDarray[0][1] = 3; //set second element of first row to 3`
- You can make as many dimensions as you like in this way, eg:
- `int complexarray[5][8][27][6]; //”4d” array`

Strings

* Now we know about general arrays, we address how C deals with strings of text.
* Text is "just" a sequence of characters, so we could just deal with strings as arrays of type `char`.
* As we deal with text a lot, C provides some additional features to make dealing with strings a little more convenient than most arrays.
* Behind the scenes, however, strings are essentially just character arrays.

Declaring Strings

* The first feature that C provides to make it easier to use strings is a special form of the initialiser for character arrays.
* A so-called *string literal* is a sequence of characters, enclosed in double-quotes (") (*not* two single-quotes (’’)).
* This is equivalent to an initialiser list containing the same sequence of characters, with an additional ’\0’ char at the end.
* The ’\0’ is used to mark the end of the string so that special string processing functions can process them properly.

```
char s[] = "A string";
/* Is the same as */
char s[] = {'A',' ','s','t','r','i','n','g','\0'};
```

`char s[99] = “A string”;` // string can be smaller than the declared array

Using Strings

* Just like other arrays, two strings cannot be operated on as a whole (only their elements can).
* C provides some functions to help manipulate strings. For example, comparing or copying strings, concatenating two strings, or finding a string length.
* All of these functions require the line:
  `#include <string.h>`
  at the start of your code

```
char str1[] = "john";
char str2[] = str1;
```

```
#include <string.h>
```

```
strcmp(str1, str2);
strlen(str1);
strcpy(str1, str2);
strcat(str1, str2);
```

Using Strings

* `strcmp` compares two strings for equality. It returns 0 if they are equal, a +ve value if `str1>str2`, and a -ve value if `str1<str2`.
* `strlen` returns the length of `str1` (the number of characters before a `'\0'` is encountered).
* `strcpy` copies the contents of `str2` into `str1`. `str1` *must be large enough to hold all of* `str2`, otherwise bad things may happen. (C doesn't check for you.)

0 if they are equal - counter intuitive?

+ve means that str1 is alphabetically AFTER str2

```
#include <string.h>


strcmp(str1, str2);
strlen(str1);
strcpy(str1, str2);
```

Using Strings

* There are many more string functions provided in string.h
* All of them, including the ones we've mentioned, rely on strings ending in `'\0'`.

Reading in Strings with `fgets()`

The standard function `fgets` can be used to read a string from the keyboard. The general form of an `fgets` statement is:

```
fgets(name, size, stdin);
```

- `name`
  is the name of a character array, aka a string variable. The line, including the end-of-line character, is read into this array.

- `size`
  `fgets` reads until it gets a line complete with ending `\n` or it reads `size` - 1 characters. It is convenient to use the `sizeof` function: `fgets(name, sizeof(name), stdin);`
  because it provides a way of limiting the number of characters read to the maximum number that the variable can hold.

- stdin
  is the file to read. In this case the 'file' is the standard input or keyboard. Other files will be discussed later under file input/output.

fgets() Example-1

Read in a string and output it's length (length.c).

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

char line[100]; /* Line we are looking at */

int main()
{
    printf("Enter a line: ");
    fgets(line, sizeof(line), stdin);

    printf("The length of the line is: %d\n", strlen(line));
    return (0);
}
```

Output:

```
Enter a line: test
The length of the line is:  5
```

test has only 4 characters - but fgets also gets the \n character.

fgets() Example-2

Read in first and last name, print out full name (full1.c).

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

char first[100];        /* first name of person we are working with */
char last[100];         /* last name                                */
char full[200];         /* full name of the person (computed)       */

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);

    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);

    printf("The name is %s\n", full);
    return (0);
}
```

`fgets()` Example-2 continued

Output of `fgets` - Example 2:

```
Enter first name: John
Enter last name: Lennon
The name is John
 Lennon
```

What happened ? Why is the last name in a new line ?

- The `fgets` command gets the entire line, including the end-of-line. For example, "John" gets stored as {'J','o','h','n','\n','\0'}.
- This can be fixed by using the statement
  `first[strlen(first)-1] = '\0';`
  which replaces the end-of-line with an end-of-string character and so end the string earlier.

fgets(); scanf(); sscanf()

```
char[30] name;
int number = 0;
printf("Enter name and phone number: ");
fgets… ?
```

`fgets()` reads the whole line as a single string. What do you do if you want to read a line in that contains more than one thing (e.g. two strings; or a mixture of ints, floats, and strings)?

Two options:

1. Use the `scanf()` function instead, which is analogous to `printf()`, and allows you to specify one or more format conversions.

2. Continue to use `fgets()` to read in the whole line but then use `sscanf()` to re-read the string (internally) and break it up into parts.

The `scanf()` Function

- The direct equivalent to the output function `printf` is the input function `scanf`. The syntax of a `scanf` statement is
  `scanf(format, &variable-1, &variable-2, ...)`
  where `format` specifies the types of variables and `&variable-1` is the address of variable-1.
- A typical `scanf` statement would be
  `scanf("%d%d%f", &a, &b, &x)`
  reading in integer values for the variables a and b and a floating point value for the variable x, entered from the keyboard.
- The `%s` conversion in `scanf` ignores leading blanks and reads until either the end-of-string '\0' or the first blank after non-blank characters. For example, if the input from the keyboard is " ABCD EFG ", `%s` will read "ABCD".

You should always specify a max length of string to read when using %s, to ensure the string fits in the variable used for it. i.e. %5s will read at most 5 characters.

Reading Numbers with `fgets()` and `sscanf()`

- Historically, scanf() has been a bit unreliable at handling end-of-lines in some implementations, and generally poor at reporting errors if it fails, so the combination of fgets() and sscanf() is often used.

Read in a number from the keyboard and double it (`double.c`).

```
#include <stdio.h>
char  line[100];    /* input line from console */
int   value;        /* a value to double */

int main()
{
    printf("Enter a value: ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &value);

    printf("Twice %d is %d\n", value, value * 2);
    return (0);
}
```

} Equivalent to: scanf(“%d” , &value);

Summary - Example

```c
#include <stdio.h>
#include <string.h>

char line[50];
char name[20];
int number;

int main() {
    puts("Enter name and number: ");
    fgets(line, sizeof(line), stdin);

    sscanf(line, "%s %d",name, &number);
    printf("\n Name and number are %s, %d \n\n",name,number);

    puts("Enter name and number: ");
    scanf("%s %d", name, &number);
    printf("\n Name and number are %s, %d \n\n",name,number);

    return 0;
}
```

What would happen if the name was longer than 19 characters?

C Lecture 4

“structured data”, Functions, and pointers

Structured Data

- Last lecture, we saw how array types let us group multiple values of the *same* type.

- Often, though, we have several pieces of data that make sense to keep together, but are of different types.

- For example:

  - name, account number, address in billing system;

  - particle x,y,z coords, vx,vy,vz coords, “particle type” (as a char) in a physics simulation.

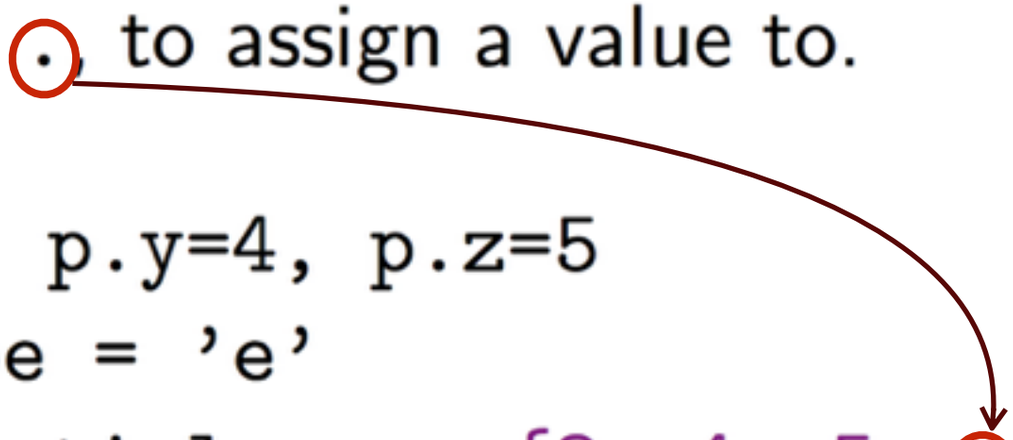
Structured data types

* `structs` (short for *structured types*) provide this functionality in C.
* We can declare a name for a `struct` type for our particle like so.
* We can then use that name to make variables with the requested internal structure, as long as we prepend `struct` to let the compiler know the context.

```
struct particle {
  double x,y,z;
  double vx,vy,vz;
  char ptype;
}
struct particle electron;
// an array of particles
struct particle p_array[5];
```

Initialising Structs

* You can initialise a struct type variable using the { } initialiser format you used for arrays.
* If we just list values, then they are assigned to the members of the struct in the order the members are defined (remaining members get set to 0).
  * You can also explicitly mention a member name, prepended with a . to assign a value to.



```
// p.x=3, p.y=4, p.z=5
// p.ptype = 'e'
struct particle p = {3, 4, 5, .ptype='e' };
```

Accessing components

We can access elements of a structured type by attaching the element name to the variable, with a joining . .

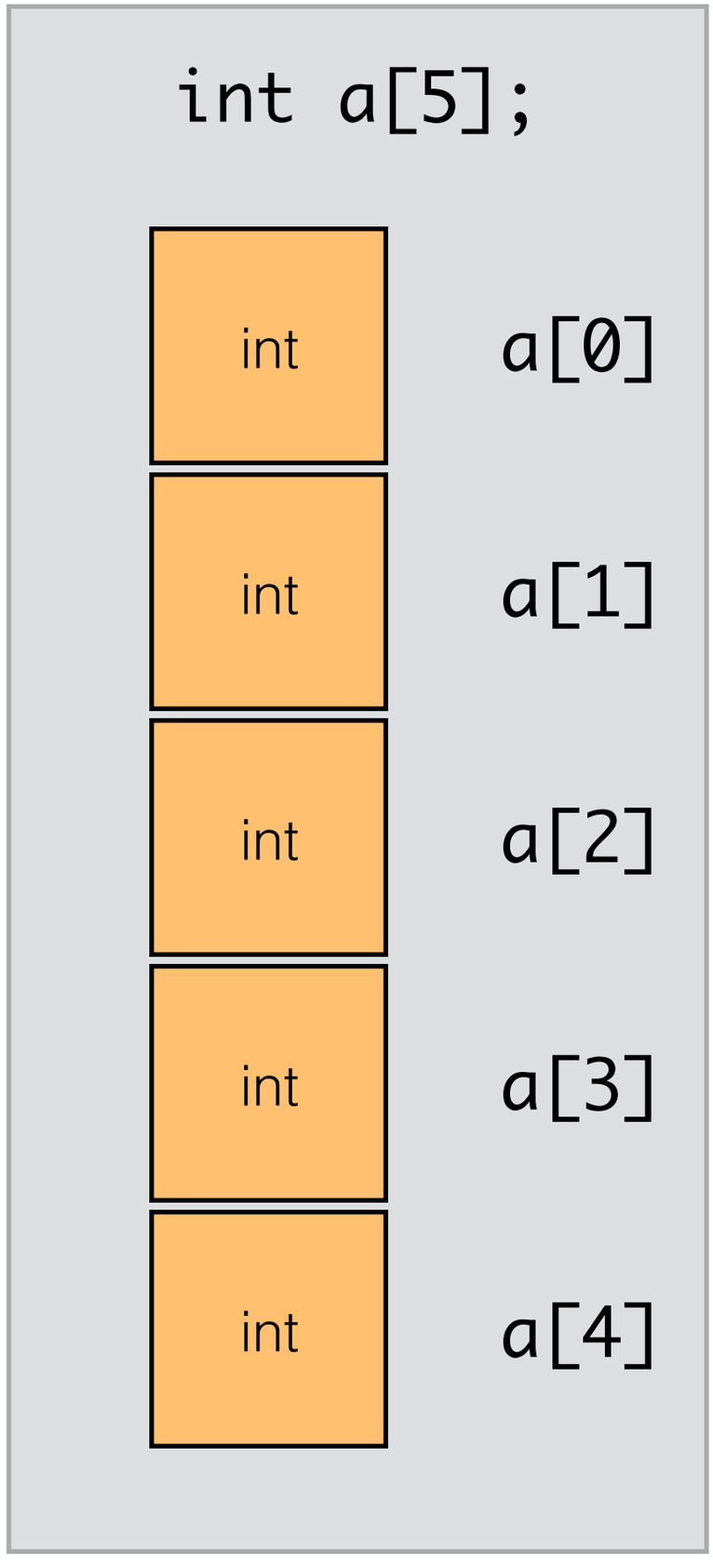
```
p.x = 3.0;
```

```
int a = p.y *2;
switch(p.ptype) {
  case 'e':
    puts("This is an electron.");
    break;
  //more code here
  default:
    puts("Unknown particle type.");
  }
```

Accessing components

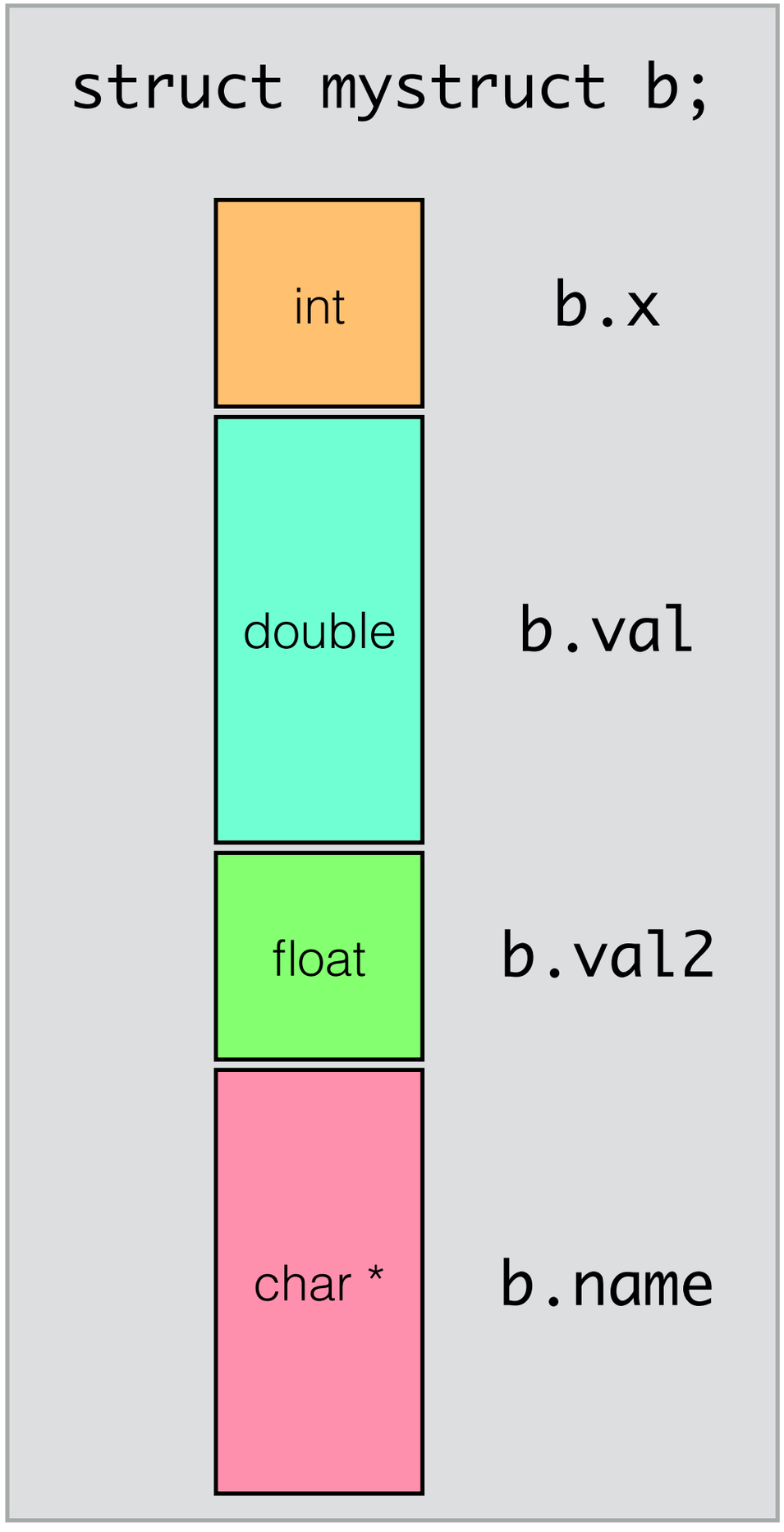
We can access elements of a structured type by attaching the element name to the variable, with a joining `.`.

```
p.x = 3.0;

int a = p.y *2;
switch(p.ptype) {
   case 'e':
      puts("This is an electron.");
      break;
   //more code here
   default:
      puts("Unknown particle type.");
   }
```



eg:

```
struct mystruct {
  int x;
  double val;
  float val2;
  char * name;
};
```



Note: this representation is purely schematic, and does not claim to represent exact layout in memory!

typedefs and structs

* The `typedef` keyword can be used to help "simplify" declaring structured types.
* `typedef` lets us specify a synonym for an existing type (including a struct type).
* If we use it for a `struct`, we can "typedef away" the need for the leading `struct` keyword when making variables of that type in future.
  * Here, we tell C that when we say "`particle_t`", we mean to say "`struct particle`".

```
//convention: end new typenames with a _t
typedef struct particle particle_t ;
particle_t electron;
```

Structs example

```c
#include <stdio.h>

struct particle {
        double x,y,z;
        double vx,vy,vz;
        char ptype;
};

typedef struct particle particle_t;

int main(void) {
        struct particle p = {1,2,3,4,5,6,'e'}; //fully initialise one particle, with full struct name

        particle_t q = {1,2,3, .ptype='e'}; //use typedef name, but only partly initialise (vx etc are 0)

        q.vx = 34.1; //set vx explicitly in q
        q.vy = 43.6;
        q.vz = q.vx/q.vy; //calculate vz using other components


        printf("x = %f\t vx = %f \t type = %c\n", q.x,q.vx,q.ptype);

        return 0;
}
```

Functions

- Often you will write a piece of code which solves a common problem.

- While *loops* let you repeat a block of code multiple times, you may want to use the same "solution" in different parts of your code, without having to rewrite it each time.

- *Functions* provide a way of "encapsulating" a chunk of code, and giving it a name so you can use it at multiple places.

- (You've already met several standard functions: `printf`, `sscanf`, `fgets` and so on.)

Function declarations

* Before we can use functions, we need to be able to declare them.
* A function declaration has two parts:
* The first part, the *function prototype*, declares the *type signature* (and type) of the function, along with its name.
* The second part, the *function body*, is a block which contains the statements that we want executed each time we call the function.

```
int f(int a);

int main(void) {
  return f(5);
}

int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```

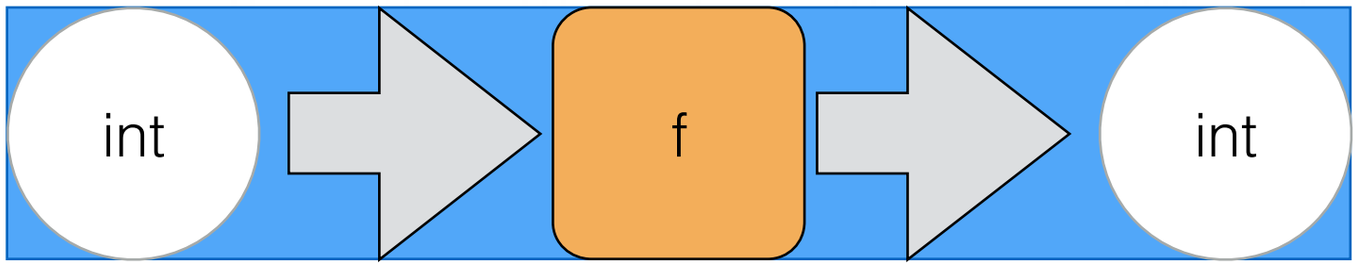
Parameters, Return types

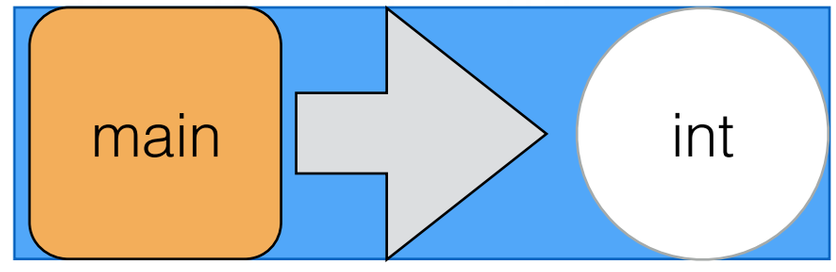
* The prototype for a function tells the compiler what values the function will need when it is invoked (the *parameters*), and what the type of the value it returns will be (the *return type*)..

```
int f(int a);

int main(void) {
  return f(5);
}
```

* In one sense, a function is like a variable with a value (the return type) that is calculated each time it is used (from some inputs).
* Compare with a mathematical function, which maps a domain to a range.

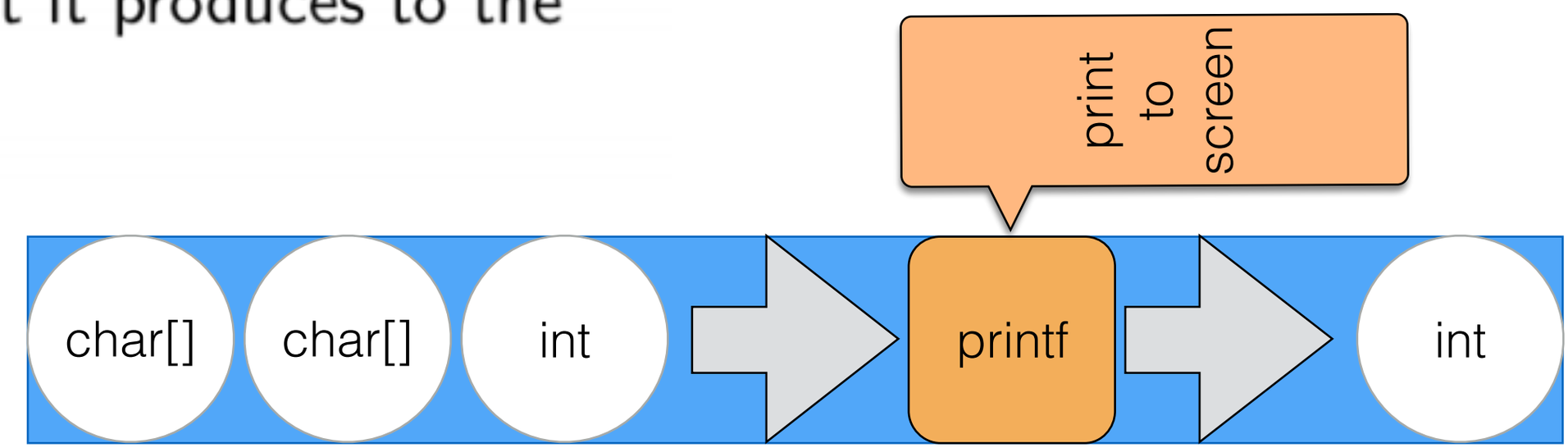
```
int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```





Side Effects

* For some functions, the most important aspect of their existence is the things they do *other* than returning a result; their *side effects*.
* `printf`, for example, is such a function: while it returns a value (the number of characters it printed), the most important thing is the output it produces to the screen.



Forward declarations

* Just as a variable declaration doesn't have to assign a value, a function prototype does not have to be followed by a function body.
* However, a function prototype for a given function does have to occur in the *file* scope, *before* the function is used in any code in the file.
* A later declaration of the function body (complete with matching prototype) must be provided.
* The "early" function prototype is called a *forward declaration*.

```c
int f(int a);

int main(void) {
  return f(5);
}

int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```

Function Definitions

* The function body is a block, with the usual scoping rules for variables declared in it.
* That is: only file scope variables, and variables defined in the body itself, are in scope.
* The parameters of a function count as variables declared in the block scope of the body.
* The values of any parameters in the function call are *copied* to the variable names in the function scope, before the rest of the function runs.

```c
int f(int a);

int main(void) {
  return f(5);
}
```

```c
int f(int a) {
  int c = a + 1;
  /* more here */
  return c;
}
```

Functions example

```c
#include <stdio.h>

// function prototype here
int f(int a); // int f(int) would be sufficient for prototype

int main(void) {
        int b = 0;
        char buffer[10];

        puts("Enter an Integer value:");
        fgets(buffer,sizeof(buffer),stdin);
        sscanf(buffer,"%d",&b);

        printf("You typed: %d \n\n",b);

        /* we can use f here, even though we've defined it later on
          as the prototype is above */
        printf("Result is: %d \n\n",f(b));

        //C passes by value, so b itself is unchanged
        printf("Value in b is still: %d\n\n",b);

        return 0;
}

//Function body here
int f(int a) { //we do need the a here - value of b is copied to a
        a += 1;
        //or a++; or a = a+1;

        return a; //this value is then the "result" of f
}
```

Pointers

- As we've just seen, functions *copy* the values in their parameters.
    - They cannot access the variable those values were provided by, directly.
- If we want a function to directly modify the contents of a variable, we need some way to tell the function about the variable itself.
- In C, we do this by telling the function "where in memory we should store values".

Pointer types

- In C, a pointer is a variable that stores the name of location in memory (an “address”).
    - Locations in memory are just assigned a number from 0 to some large value.

- Pointers have types, which tell the compiler how we’d like to interpret the value at that address - is it an `int`, a `double`, or whatever.

Declaring pointers

* Much as with arrays, we declare a pointer to a type by "decorating" a name with a symbol.
* For arrays, we had to add [ ] to a name to make it an array of values of that type.
* For pointers, we add a * to the start of the name.
* Here `x` and `y` are variables of type `int`. `p` is a variable of type *pointer to int*.

```
int x, *p, y;
```

NULL

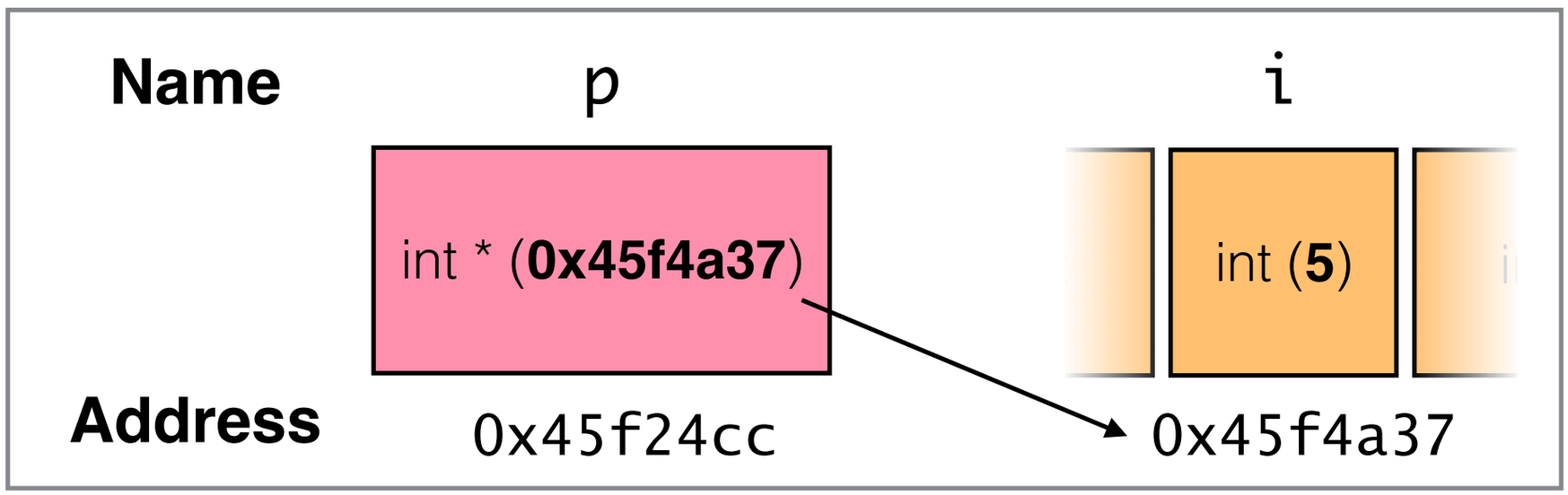
* We can give a pointer variable an initial value, but unlike value types, it is very hard to provide a value that will be useful.
* Clearly, we shouldn't point a pointer at an area of memory that *doesn't* contain a value of the right type!
* The most useful literal pointer value is the special value `NULL`.
* A pointer with the value `NULL` is essentially set to "not point at anything"

```
int x, *p, y;
/*This pointer is
pointing at nothing
explicitly */
int *p1 = NULL;
```

Pointing at a variable

* We will almost always want our pointers to point at the memory used by an existing variable.
* The special operator & provides the address in memory where a variable is located.

```
int i = 5;
//Give p the location where i is currently stored.
int *p = &i;
```



Getting a value from a pointer

* We will almost always want our pointers to point at the memory used by an existing variable.
* The special operator & provides the address in memory where a variable is located.
* The special operator * does the opposite, providing the value located at the address in the pointer.

```
int i = 5;
//Give p the location where i is currently stored.
int *p = &i;
//Compare value in i to value in location p points at.
return (i == *p);
```

Passing pointers to functions

* So, now we can use the & and * operators to allow a function to modify a variable (rather than just use the value in it)
* Here, the function `f` takes a *pointer to an integer* as its argument.
* It then modifies the value in the location the pointer points at, by adding 2 to it.
* In `main`, we call `f` with the *address* of `i`, so `f` modifies `i`'s *contents*.
* The value printed is 9, (not 7).

```
void f(int * p)
{
   *p += 2;
}
int main(void){
   int i = 7;
   f(&i);
   printf("%d",i);
}
```

Pointers example

```c
#include <stdio.h>

int main(void) {
        int i = 6;

        /*
                We can either declare and assign a pointer in one go:

                int *p = &i;

                or in two steps:

                int *p = NULL; //declare pointer to int, safely set to NULL
                p = &i; //and assign it the address of i

        */

        int *p;
        p = &i;
        printf("\n value of i is %d. Value in location %p is %d\n\n",i,p,*p);

        return 0;

}
```

Pointers example 2

```c
#include <stdio.h>

/* Here is a new function prototype */

void f(int * a_ptr); //void f(int *) would also be sufficient

int main(void) {
        int b = 0;
        char buffer[10];

        puts("\n Enter an integer");
        fgets(buffer,sizeof(buffer),stdin);
        sscanf(buffer,"%d",&b);

        f(&b); //call f with b's address

        printf("Value in b is now: %d \n\n",b);

        return 0;
        }

void f(int * a_ptr) {
        //explicitly modify value in memory at a_ptr
        *a_ptr += 1;
        //equiv to (*a_ptr)++; or *a_ptr = (*a_ptr) +1;
        //need brackets to ensure we don't actually increment a_ptr itself.
}
```

C Lecture 5

More pointers, Command Line Arguments,
C Build Process (1)

Beware Pointers!

- Last lecture, we saw how pointers could be used to allow you to access memory directly.
- This is useful for allowing functions to directly modify the contents of variables whose names are out of scope…
- …but it's also dangerous, as we have to be sure that the memory we're pointing at is still used for that variable.
- This requires a bit of understanding of how memory for variables is allocated, by default.

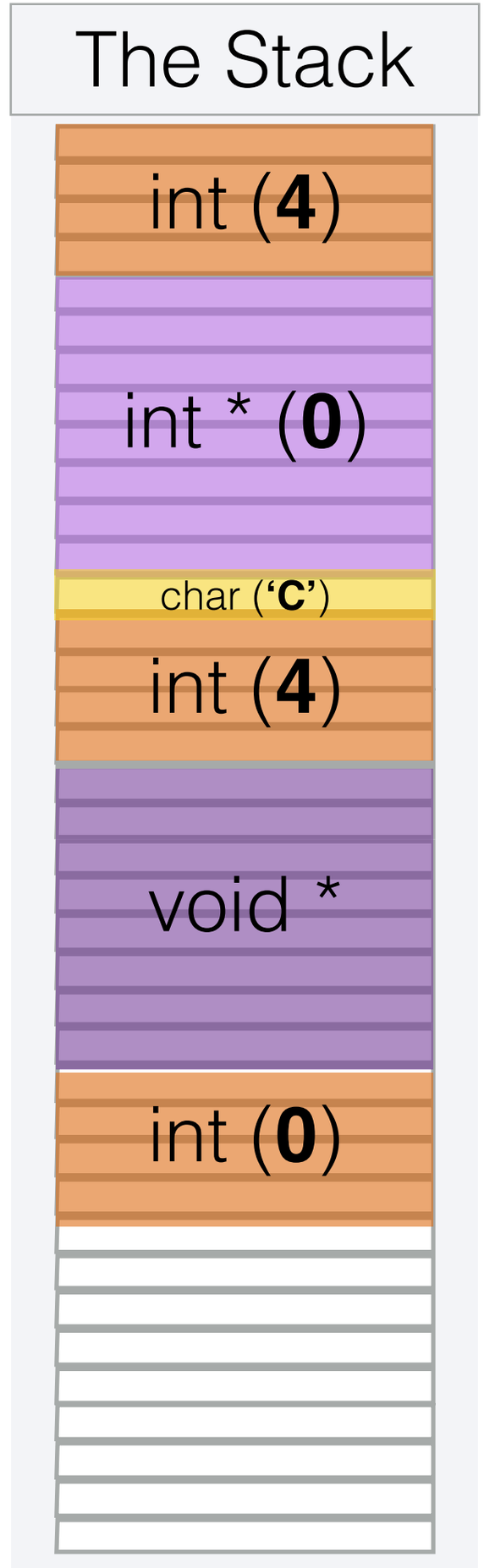
Automatic allocation

```c
#include <stdio.h>

int * f(int, char);

int myfunc(int);


int main(void){

        const int a_number = 4;
        int * ptr = NULL;
        ptr = f(a_number,'C');
        printf("Ptr points to: %p, value %d\n", ptr, *ptr);
        printf("Value in ptr is now: %d!\n", *ptr);
        return 0;
}

int * f(int a, char letter) {
        int b = 0 ;

        //yes, you can do this
        //chars are just a numeric index
        b = letter + a;
        return &b;
}
```

(This representation is purely schematic, and simplifies the actual stack and call structure.)

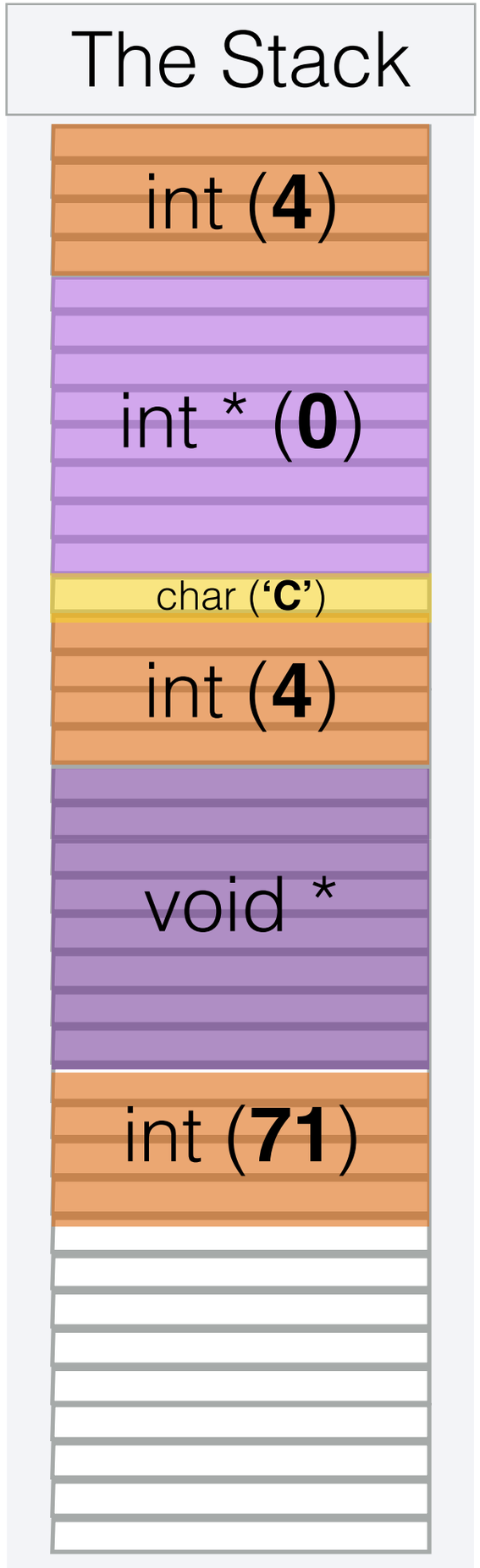
Automatic allocation

```c
#include <stdio.h>

int * f(int, char);

int myfunc(int);


int main(void){

        const int a_number = 4;
        int * ptr = NULL;
        ptr = f(a_number,'C');
        printf("Ptr points to: %p, value %d\n", ptr, *ptr);
        printf("Value in ptr is now: %d!\n", *ptr);
        return 0;
}

int * f(int a, char letter) {
        int b = 0 ;

        //yes, you can do this
        //chars are just a numeric index
        b = letter + a;
        return &b;
}
```



Automatic allocation

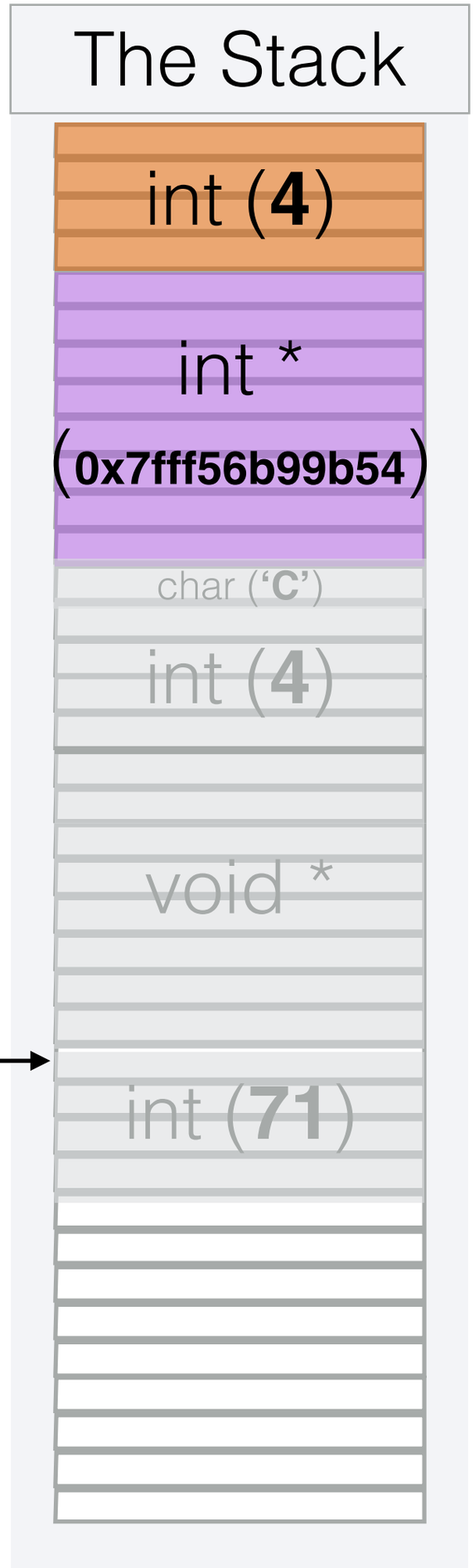
```c
#include <stdio.h>

int * f(int, char);

int myfunc(int);


int main(void){

        const int a_number = 4;
        int * ptr = NULL;
        ptr = f(a_number,'C');
        printf("Ptr points to: %p, value %d\n", ptr, *ptr);
        printf("Value in ptr is now: %d!\n", *ptr);
        return 0;
}

int * f(int a, char letter) {
        int b = 0 ;

        //yes, you can do this
        //chars are just a numeric index
        b = letter + a;
        return &b;
}
```



Automatic allocation

```c
#include <stdio.h>

int * f(int, char);

int myfunc(int);


int main(void){

        const int a_number = 4;
        int * ptr = NULL;
        ptr = f(a_number,'C');
        printf("Ptr points to: %p, value %d\n", ptr, *ptr);
        printf("Value in ptr is now: %d!\n", *ptr);
        return 0;
}

int * f(int a, char letter) {
        int b = 0 ;

        //yes, you can do this
        //chars are just a numeric index
        b = letter + a;
        return &b;
}
```

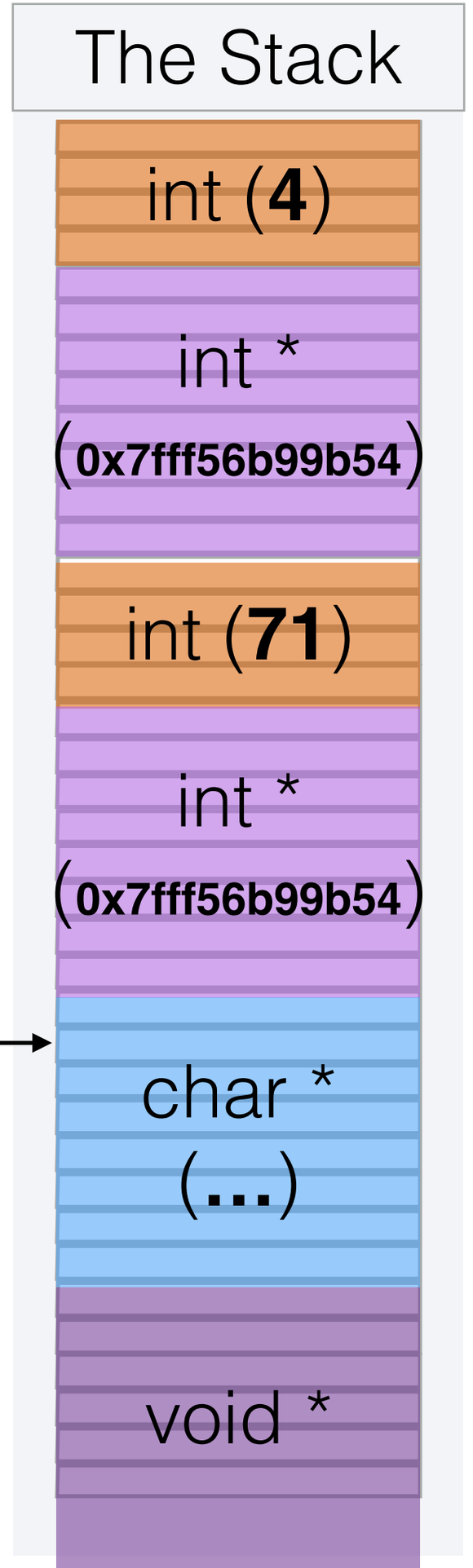
0x7fff56b99b54



Automatic allocation

```c
#include <stdio.h>

int * f(int, char);

int myfunc(int);


int main(void){

        const int a_number = 4;
        int * ptr = NULL;
        ptr = f(a_number,'C');
        printf("Ptr points to: %p, value %d\n", ptr, *ptr);
        printf("Value in ptr is now: %d!\n", *ptr);
        return 0;
}

int * f(int a, char letter) {
        int b = 0 ;

        //yes, you can do this
        //chars are just a numeric index
        b = letter + a;
        return &b;
}
```

0x7fff56b99b54



Pointers to structs

- We can create pointers to structured data types, just as we can to basic types.

- Care is needed when accessing components of the value.

Dereference ptr before taking component of value

-> does both the dereferencing and the component selection in one go.

```
struct particle *muon; //a pointer to a struct
(more code here)

(*muon).x = 3;

or

muon->x = 3;
```

Pointers and arrays

- We can also create pointers to arrays… except that it turns out, we don't need to!

- The "base name" of a array variable can be treated as a pointer by C, and assigned to pointer variables.

C will convert "array style" and "pointer style" accesses.

```
int a[5]; //an array of 5 ints
int *p = a; //p now points at the start of a

*p = 3; //a[0] is now 3!

p[2] == a[2] //true!

Or

*(p+4) == a[4]; //also true, using ptr arithmetic
```

Arrays and Functions

- When you pass an array to a function, C actually passes a pointer.

- This means that modifying the array in a function *does* modify the same array you passed to it.

- However, the function cannot know the "size" of the array you passed it (it just gets a pointer). You must also pass the size of the array to a function to let it know how big the array is.

```
int f(int n, int a[]);

int array[5];

f(5,array); //pass size appropriately
f(sizeof(array)/sizeof(array[0]),array); //or this
```

sizeof(a) in f is the size of a pointer, not the size of array!
sizeof(a[0]) *is* the size of an int, still!
We use n in any loops in the function that need size of a.

Multidimensional Arrays

- For multidimensional arrays, things are a bit more complex. (But less complex with C99!)

- The function will get passed a pointer to an array of one less dimension. So, the function will know about all the sizes except the left-most dimension.

- (You *must* still specify the sizes yourself so the function can work out what those dimensions should be!)

C99: You can use variables to specify an array dimension

```
int f(int n, int m, int a[][m]);

int array[5][3];

f(5, 3, array); //pass size appropriately
```

sizeof(a) in f is the size of a pointer, not the size of a!
sizeof(a[0]) *will* be the size of an array of 3 ints!
sizeof(a[0][0]) *will* be the size of an int!

Early exit of a program.

- While return allows you to return a value from a function, exiting the function itself…
- …it doesn't let you end the entire program. (return in main, of course does this as a side effect!)
- If you ever need to stop a program suddenly (usually because something has gone wrong), you can use the `exit` function, which is defined in `stdlib.h`

```
exit(1);
```

exit the program *right this moment*, with a return code set to 1

Command line arguments

* Now that we know more about how to write functions, and about pointers...
* The function `main` is special in that we can declare it in two different ways.
* The first signature, `int main(void)`, is what we've been using up to now.
* The second signature, `int main(int argc, char * argv[])`, adds two parameters to our function.
* If we declare `main` like this, then the compiler will use the parameters to tell us about arguments passed to the program when it is executed on a command line.

Command line arguments

* The value of `argc` will be the number of arguments passed to the program (plus 1, as the first "argument" is always the name of the program).
* `argv` is an array of strings that contain, in order, the text of each argument.
* (`argv` is of type `char * []` and not `char[][]` because the different arguments will not all have the same length, and a multidimensional array must be square. Instead, we use the array/pointer magic that C provides to access the char * like separate arrays.)

Compare argc to $# in bash, argv to $@ in bash.

Command line arguments example

```c
#include <stdio.h>

int main(int argc, char * argv[]) {

        int tmp, result = 0;

        if (argc != 4) {
                puts("Please provide exactly 3 arguments!");
                return 1; //non-zero error code - see Gareth's slides
        }

        for(int i=1; i<4; i++) {
                sscanf(argv[i], "%d", &tmp);
                result += tmp;
        }

        printf("The sum of the numbers is: %d\n", result);

        return 0; //zero error code, as success!

}
```

```
gcc -Wall cla.c
./a.out 1 2 4

The sum of the numbers is: 7
```

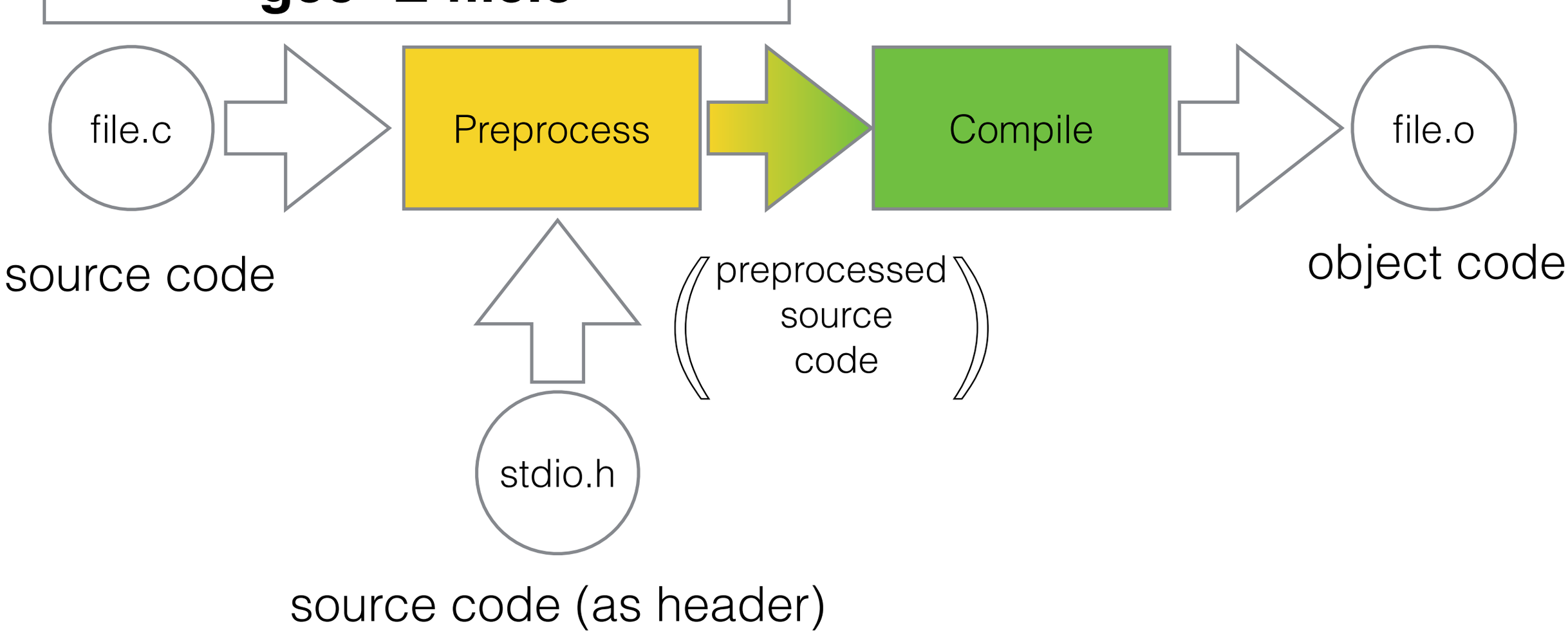
Building Multi-file projects

- Up to now, all the code we've written has been in a single file.
- Managing a large project like this quickly becomes unmanageable.
- We need to know how to combine code in multiple source files into one program.
- In order to do this, we also need to understand more deeply how C code is taken from source to final executable.

The C Build Process

gcc -c file.c

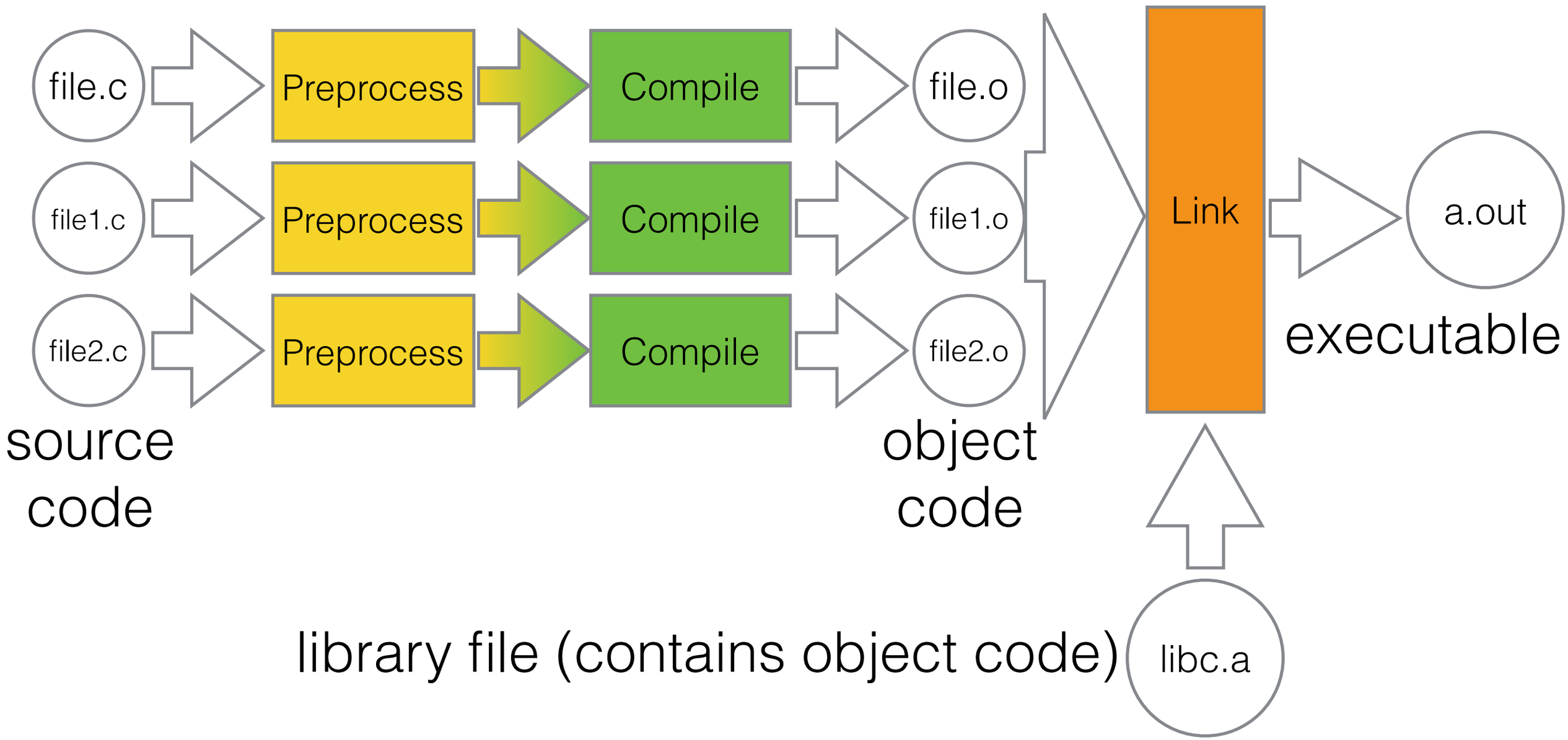
gcc -E file.c



The C Build Process

gcc file.c file1.c file2.c

gcc file.o file1.o file2.o



The Preprocessor

- The preprocessor "prepares" the initial source code for the compiler.
- The output is *also* C source code.
- The preprocessor performs text manipulation on the input source
  - replace text with other text
  - include text (based on a test)
  - insert text from another file at this point
- All *preprocessor directives* start with a #

#define

- `#define` is the simplest preprocessor directive.
- It takes a single "macro", and replaces it with the provided text, everywhere in the rest of the file.
- (Text inside "" is not replaced.)

```
#define APPLE DELICIOUS FRUIT

This is an APPLE, not a SAPPLE.
"Quoted APPLE text."
```

preprocess

```
This is an DELICIOUS FRUIT, not a SAPPLE.
"Quoted APPLE text."
```

Doing this with numeric literals is one way to set global constants, like π.

```
#define PI 3.14159
```

"Predefined" macros

- C language provides some macros which will automatically be replaced with useful values by the preprocessor.
- `__LINE__` will be replaced with the line number it appears on.
- `__FILE__` is the name of the file being processed.
- `__DATE__` and `__TIME__` are replaced appropriately.

```
printf("We are on line %d in file %s.\n", __LINE__, __FILE__);
```

```
printf("We are on line %d in file %s.\n", 23, "myfile.c");
```

#ifdef … #else … #endif

- User defined macros can also be used to control if a piece of text is kept in, or removed from, the source code, using `#ifdef` constructs.

```
#define APPLE

#ifdef APPLE
We know about apple.
#else
We don't know about apple.
#endif
```

APPLE is now #defined, albeit to an empty value.

or

```
#define APPLE

#ifndef APPLE
We don't know about apple.
#else
We know about apple.
#endif
```

preprocess → `We know about apple.` ← preprocess

If APPLE wasn't #defined, what would the resulting text be?

#include

- `#include` copies all the text from the named file, and inserts it into the text at the point of the directive. (*The preprocessor will then process the included text, if it also contains directives.*)

```
#include "mytext"

This is also some BANANA
```

mytext

```
This is the text in mytext.
It is not very interesting.
#define BANANA text.
```

preprocess

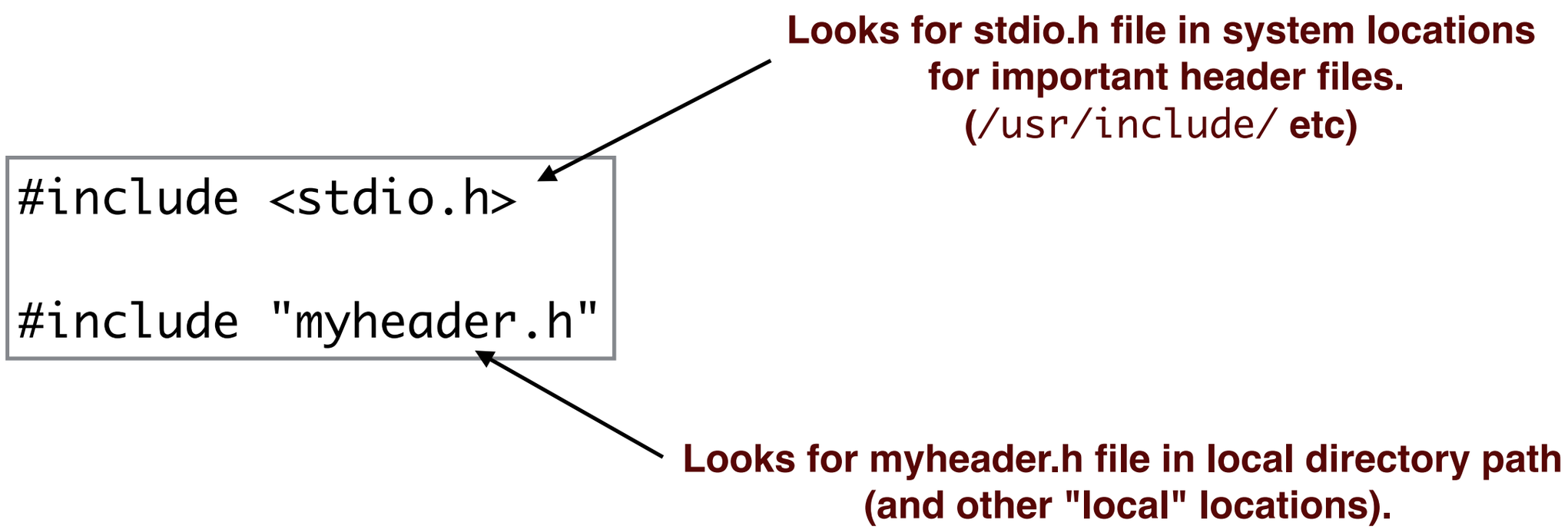
```
This is the text in mytext.
It's not very interesting.

This is also some text.
```

`#define` **in mytext makes this change here**

#include <> versus ""

- If you specify the filename in an `#include` like `<stdio.h>` then the preprocessor will look for the file in some standard system locations.

- If you want to refer to a file that's in the same directory as your file (for example), then use `"myfile"`.



Common definitions

- In order to allow code in one (source) file to know about functions and values in other (source) files, we need a way to add:
  - function prototypes for functions defined in other files
  - common constants etc
- We can then use those functions in our other files (and let the linker join up all the code at the end).

Header files

- C calls a file which exists just to be `#include`d to provide common declarations a "header file".
- Conventionally, they have the `.h` suffix (but they contain normal C code).
- For every `.c` file you write, which contains code you want to reference in other `.c` files:
  - Make a `.h` file for the prototypes etc you want to reference elsewhere.

#include guards

- It's an error in C to declare a function (or any other name) more than once in the same file.
- If our definition is in a header, how do we stop the header being included more than once?

file.h

```
#ifndef FILE_H
#define FILE_H

(text of header)

#endif
```

The first time this is `#include`d, the `#define` activates, and defines FILE_H. The *second* time we include it, the `#ifndef` skips the entire file!

Modern compilers also support putting `#pragma once` at the top of a header file, for same effect. (This is *not* universally supported!)

Preprocessing example

example.c

```c
#include <stdio.h>
#include "function.h"

int main(void){

        printf("The result of function f is: %f\n", f(27));

        return 0;
}
```

What happens if we don't include function.h?
(What does the warning mean?)

function.h

```c
//declaration of f (multiplies number by 2)
double f(double);
```

function.c

```c
double f(double a) {
        return a*=2;
}
```

```
gcc -E example.c | less
```

(what is the prototype for printf?)

The Compiler

- The compiler takes C source code and translates it into actual machine code.
- Source code is broken down into syntactic units (“tokens”), and the relationships between them.
- Instructions are then converted to machine code, with “symbols” attached to named items for reference.
- The resulting machine code, annotated with symbols, and prepended with a list of all the symbols in the file, is called “object code”.

Optimisation

- As with natural languages, naïve compilation (translating "word for word" into machine code) doesn't always give the best result.

- We can ask the compiler to spend more time and effort to produce more efficient, or faster, or shorter, representations - *optimisations*.

- We specify these with the `-O` flag, specifying a value from `0` (no optimisations) through `3` (all optimisations).

- In general, `-O2` is a good balance between compilation time and code performance.

Compilation example

example.c

```c
#include <stdio.h>
#include "function.h"

int main(void){

        printf("The result of function f is: %d\n", f(27));

        return 0;
}
```

function.h

```c
//declaration of f (multiplies number by 2)
int f(int);
```

function.c

```c
int f(int a) {
        return a*=2;
}
```

```
gcc -c example.c
gcc -c function.c

nm example.o
nm function.o
```

Further reading

- You can read more about the preprocessor (and more features which are not covered in the course), here:

- https://gcc.gnu.org/onlinedocs/cpp/

- You can read a bit more about compiler optimisation here:

- http://moodle2.gla.ac.uk/mod/page/view.php?id=149957

C Lecture 6

C Build Process (2) [Linking and Libraries], File I/O

The Linker

- The Linker is responsible for joining up the object code from the Compiler, so that all symbols are mapped to their representation.
- Firstly, it takes all of the object code files it is given (one for each C source file), and turns them into one large file, with “consolidated” symbol index at the top.
- Before object code is linked, function calls are simply a note of the symbol that corresponds to the function they want to call.
- The linker looks up the symbol in the symbol index and replaces it with a call to the function code with the corresponding symbol.
- This also happens for variable names in the file scope in each file.

Libraries

- One way to provide a set of useful functions to others is to package them all up into a "library".

- This is done with a linker (but we won't cover how in this course)

- This is just a specially packaged set of object code (with an index to make it easy to find things in it).

- Libraries have names ending in `.a` or `.so` (the difference is in how and when they are linked).

- We can ask the linker to link to a library called *libname.a* (or *libname.so*) with the option `-lname`

- In this case, the linker will also match symbols in our code with symbols in the library mentioned.

The C Standard Library

- One library is used in (almost) all C code you will ever write: `libc.a` (or `libc.so`).

- The "C Standard Library" contains useful functions for a large number of potential applications, including I/O etc.

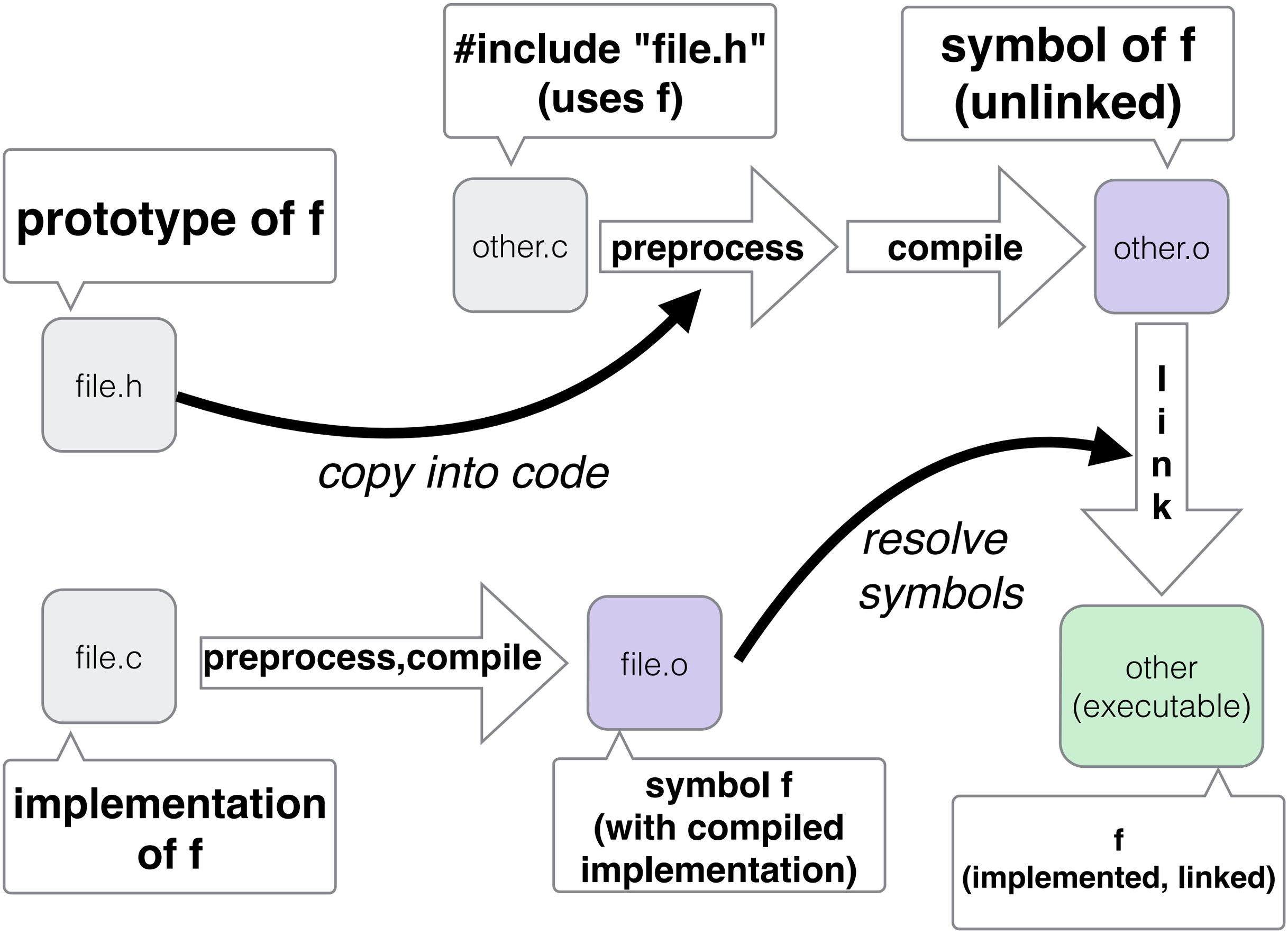
- It's so large that the prototypes for its functions are split into multiple header files, grouped by topic.

  - `stdio.h` provides prototypes for the I/O functions in libc

  - `string.h` provides prototypes for the string functions in libc

- As libc is so commonly used, the linker will link it without even being asked.

An exception is the (floating point) mathematics part of libc. For historical reasons, this is often stored in a separate library libm.a / libm.so , which *does* need explicitly linked.


prototype of f
file.h
#include "file.h"
(uses f)
other.c
preprocess
compile
symbol of f
(unlinked)
other.o
copy into code
resolve
symbols
link
file.c
preprocess,compile
file.o
implementation
of f
symbol f
(with compiled
implementation)
other
(executable)
f
(implemented, linked)

prototype of f

file.h

#include "file.h"
(uses f)

other.c

preprocess

compile

symbol of f
(unlinked)

other.o

l
i
n
k

copy into code

resolve symbols

file.c

file1.c

file2.c

preprocess, compile

preprocess, compile

preprocess, compile

file.o

file1.o

file2.o

link

libstuff.a
or
libstuff.so

symbol of f
(implemented, in library)

other
(executable)

f
(implemented, linked)

Libraries and Linking example

example.c

```c
#include <stdio.h>
#include "function.h"

int main(void){

        printf("The result of function f is: %d\n", f(27));

        return 0;
}
```

function.h

```c
//declaration of f (multiplies number by 2
int f(int);
```

function.c

```c
int f(int a) {
        return a*=2;
}
```

```
gcc example.c function.c
```

or (if .o files exist already)

```
gcc example.o function.o
```

```
nm a.out
```

(where does printf symbol get resolved?)

```
ldd a.out
```

Libraries and Linking example

example.c

```c
#include <stdio.h>
#include "function.h"

int main(void){

        printf("The result of function f is: %d\n", f(27));

        return 0;
}
```

Need to explicitly link libm.a/libm.so
(name is always the bit between "lib" and .a/.so)

```
gcc example.c function2.c -lm

ldd a.out
```

function2.h

```c
//function returns cosine of argument
double f(double);
```

function2.c

```c
#include <math.h>

double f(double a){
        //cos function declared in math.h, implemented in libm.a
        return cos(a);
}
```

mandel.c

complex.h header for complex numbers
math.h for complex math functions
png.h for writing png image
stdint.h for "exactly 8 bit wide" ints
initpng.h for own code to initialise a png

```c
#include <stdio.h>
#include <complex.h>
#include <math.h>
#include <png.h>
#include <stdint.h>
#include "initpng.h"

#define DIM 500

uint8_t mandel_kernel(double complex c);

int main(void) {

        uint8_t array[DIM][DIM];
        double complex c;

        for(int i=0; i<DIM;i++){
                for (int j=0;j<DIM;j++){
                        //align on range -2..2 in x and y
                        c = (4*(double)i/DIM)-2 + ((4*(double)j/DIM)-2)*I;
                        array[i][j] = mandel_kernel(c);
                }
        }

        FILE *fp = fopen("mandel.png", "wb");

        png_structp png = initpng(fp,DIM);

        uint8_t * rows[DIM];
        for(int i=0;i<DIM;i++) {
                rows[i] = (uint8_t *)array[i];
        }

        png_write_image(png, rows);
        png_write_end(png, NULL);

        fclose(fp);

}
```

exactly 8 bit wide integer

floating point complex number!

get "rows" of array, as ptrs

mandelbrot set calculation

```c
uint8_t mandel_kernel(double complex c) {
        uint8_t i = 100;
        double complex z = 0 + 0*I;
        while(cabs(z)<2.0 && i>0){
                z = cpow(z,2) + c;
                i++;
        }
        return i;
}
```

cpow raises z to the power 2 (efficiently)
cabs gives the absolute value of z

```
gcc mandel.c initpng.o -lm -lpng
```

link to libm.so (for math)
link to libpng.so (for png creation)

References

- The documentation for the (GNU) implementation of the C Standard Library:

- https://www.gnu.org/software/libc/manual/

- (We have not covered a lot of the contents!)

File I/O

- To date, we've used various I/O functions without really exploring where they come from.

- We're going to take a slightly more in-depth look at how C does input and output here.

- Our context is "writing and reading to files"…
    - but *everything* in Linux is a file, so this also applies to the other I/O we've done.

- All I/O functions are declared in `stdio.h` (and implemented in libc)

"Stream I/O"

- C provides several ways to think about I/O. In this course, we will consider two kinds, both "stream I/O" models.

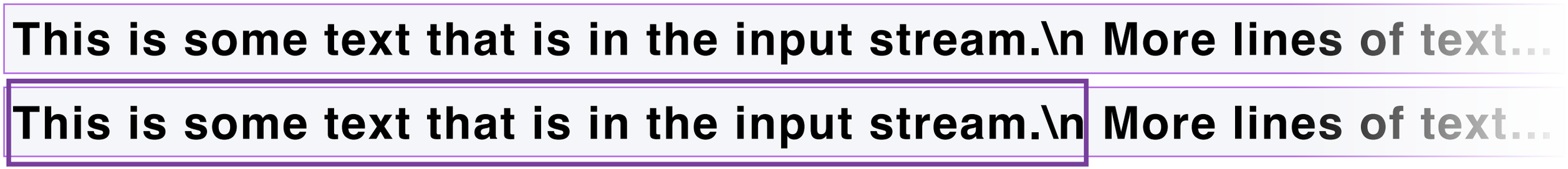
- Stream I/O considers everything as a "stream" of "characters".

- Functions can take some characters out of a stream (reading).

- Functions can also put some characters into a stream (writing).

- A stream can represent any file (or file-like thing - such as the special STDIO, STDIN, STDERR files for handling user input/output).

Text Stream I/O



fgets(…)

fgets takes characters from the stream,
until it reaches a newline

After fgets returns, the stream no longer contains the characters it took.

(If the stream represents a file, then the file itself isn't changed - we've just "moved on in the file" past those characters.)

ASCII

- The "characters" in the stream are encoded in the format known as ASCII (American Standard Code for Information Interchange).

- This is also what is used for your `char` variables.

- It assigns values from 0 to 127 to different symbols (including the Latin letters, lowercase and uppercase).

ASCII cannot represent non-Latin alphabets. More modern languages than C use other codes - Unicode, for example - which can represent many more characters, alphabets, etc.

NULL = \0 character

space character

newline character

ASCII Symbol Table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	(NUL)	(SOH)	(STX)	(ETX)	(EOT)	(ENQ)	(ACK)	(BEL)	(BS)	(HT)	(LF)	(VT)	(FF)	(CR)	(SO)	(SI)
(16)+	(DLE)	(DC1)	(DC2)	(DC3)	(DC4)	(NAK)	(SYN)	(ETB)	(CAN)	(EM)	(SUB)	(ESC)	(FS)	(GS)	(RS)	(US)
(32)+	(SP)	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
(48)+	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
(64)+	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
(80)+	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
(96)+	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
(102)+	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	(DEL)

Codes in () are non-printing characters (most of them were originally designed to control automatic typewriters). The relevant ones for our use are indicated.

Opening a File

Name of file to open

- `FILE * fopen(char[] filename, char[] mode)`

- A `FILE` is a special structured type which represents the stream of data flowing to or from a file.

- fopen will try to open a file with the path and name we specify, and return a pointer to a `FILE` representing it.

- We also need to specify how we are going to access the file.

The special file pointers STDOUT, STDIN and STDERR are automatically opened when a program starts and correspond to the same concepts you met in the Bash part of the course.

File access modes

File mode	Intent	Equivalent Bash operator	Effect of adding a +
`"r"`	Just read the file	<	`"r+"` - read and write the file (from the start)
`"w"`	Just overwrite the file	>	`"w+"` - overwrite the file, but allow reading too
`"a"`	Just write stuff to end of file	>>	`"a+"` - start writing to end of file (but allow reading too)

```
FILE * fp = fopen("myfile.txt", "r");
```

Reading from a File

- `char * fgets(char * string, int len, FILE * file);`
- `int fscanf(FILE * file, char * format, …)`

Any number of pointers to variables to match format

- fscanf returns an int, which is the number of variables it successfully put values into.
- (This can be less than the number requested, if it was unable to interpret some of the stream in the requested way.)
- fscanf returns `EOF` if they reach the end of a file (and therefore there's nothing more to read).

fscanf(stdin, "%d is an integer\n", &myint) is identical to scanf("%d is an integer\n", &myint)

fscanf

900 6 8.000\n980 h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n

900 6 8.000\n980 h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n

fscanf(fp,"%d %d %f\n",…)

format matches next characters in stream
(fscanf returns 3)

980 h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n7 34 4.4359\n

fscanf(fp,"%d **%d** %f\n",…)

format can't match second conversion specifier
('h' is not interpretable as an int)

h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n7 34 4.4359\n789

stream is left at point of first "non-matching" characters.
fscanf returns 1; first variable (pointer) passed to fscanf is assigned value 980 (as an int), the other two are *unchanged*.

Writing to a File

- `int fputs(char * string, FILE * file);`
- `int fprintf(FILE * file, char * format, …);`

Any number of values to match format

- fputs and fprintf both return an int - for fprintf, this is the number of characters it wrote.
- If they fail to write to the file, they instead return the special value `EOF`.
- Unlike puts, fputs does not add a '\n' to output.

```
fprintf(file, "%d is an integer\n", myint);
```

fprintf(stdout, "%d is an integer\n", myint) is identical to printf("%d is an integer\n", myint)

Streams and Buffers

- Strictly, writes to a stream are not immediately reflected in the file they represent.

- (Physical media like hard disks, or optical disks, takes real time to write stuff.)

- Instead, the writes are "queued up" in a buffer of things needing to be written.

- Periodically, the buffer is emptied, and its contents actually committed to the file.

Closing a File

- `int fflush(FILE * fp);`

- fflush makes sure that all the data we've written so far has actually been committed to the file itself (it flushes the buffer immediately).

- `int fclose(FILE * fp);`

- fclose does a flush, and then removes the connection between fp and the file it's attached to. If successful, it returns 0.

- After closing a file, the file pointer *cannot* be used for I/O until assigned a new file with an fopen.

NEVER fclose stdin,stdout or stderr! (fflush can be useful to ensure stuff is printed immediately).

General I/O Functions

- In general, I/O functions are written in the form objectverbsubject.
- Where object can be **f**, for files, **s**, for strings, or omitted, for “default/terminal”.
- (There are other options, such as **sn** for ”string, along with a max length”, but we leave those for the documentation.)
- And subject is **f** for “formatted values”, **s** for “to string”, or **c** for “single character”.
- sprintf, e.g., uses a format string to convert values to text, and stores the result in a string.
- (There are some violations of these general rules, especially for the get and put families, and some combinations do not exist, so check the documentation before using.)

Stream I/O Example

```c
#include <stdio.h>

int main(void){

        FILE * fp = fopen("testfile.txt","w");

        fprintf(fp, "Number %d, %f", 5, 5.0);

        fflush(fp); //ensure everything written to file
        fclose(fp); //although fclose would do this for us

        fp = fopen("testfile.txt", "r");

        int a;
        double b;

        fscanf(fp, "Number %d, %lf", &a, &b);

        fclose(fp);

        printf("Double in file was %f\n", b);

        return 0;

 }
```

(Strictly unneeded here, as we fclose immediately after)

(ASCII) text representation of text and numeric values

```
hexdump -C testfile.txt
00000000  4e 75 6d 62 65 72 20 35  2c 20 35 2e 30 30 30 30  |Number 5, 5.0000|
00000010  30 30                                             |00|
00000012
```

text '5' not value 5

text '5.000' not double

Binary Stream I/O

- Rather than opening a file as if it were text, we may want to access the bytes in a file directly.
  - This can be more efficient than text stream I/O
  - Also doesn't lose precision (consider floating point values).

I/O type	Text Stream	Binary Stream
Pros	Human readable	Exact representation of variables
Cons	Precision loss Often larger than needed	Opaque Less portable

Opening, Closing Binary Streams

- We can still use fopen and fclose to work with Binary Stream I/O.

- There's just one change: the file mode must have a "b" added to it, to ensure we directly read precisely what's in the file.

- So, "r" -> "rb", "w+" -> "wb+" and so on.

```
FILE * fp = fopen("myfile.txt", "rb");
```

The C standard allows text streams to do "conversion" between the file and the stream itself. On Linux (and other POSIX) systems, no such conversion occurs, but other platforms, including Windows, do. For portability, *always* explicitly turn on binary mode for binary I/O.

Reading, Writing Binary

- `long fread(void * start, long size, long num, FILE * file);`
- `long fwrite(void * start, long size, long num, FILE * file);`

•fread reads `size*num` bytes from `file`, and inserts them into memory starting at `start`. (It returns how many it actually wrote).

•fwrite does the same thing, but takes *from* `start`, and writes *into* `file`.

```
int a[5];
fread(a,sizeof(int),5,fp);
```

Binary Stream I/O Example

```c
#include <stdio.h>
#include <string.h>

int main(void){

        char str[] = "Example string";

        int a[5] = {0,1,2,3,4};

        FILE * fp = fopen("testfile.bin","wb");

        fwrite(str,sizeof(char),strlen(str), fp);
        fwrite(a,sizeof(int),5, fp);

        fclose(fp);

        return 0;
}
```

Binary version of text… is still text

```
 hexdump -C testfile.bin
00000000  45 78 61 6d 70 6c 65 20  73 74 72 69 6e 67 00 00  |Example string..|
00000010  00 00 01 00 00 00 02 00  00 00 03 00 00 00 04 00  |................|
00000020  00 00                                            |..|
```

Direct binary values of int type (don't look like text)

C Lecture 7

Memory allocation
Pseudo Random Number Generation

Teaching Week	Week Beginning	Mon 2-5pm	Tue 2-3pm	Wed 2-5pm	Thu 2-3pm	Fri 2-5pm
21	04-Jan-16					
22	11-Jan-16		Intro C/Linux		C-1	
23	18-Jan-16		Linux-1	Lab-0	Linux-2	Lab-0
24	25-Jan-16	Lab-0	C-2	Lab-1		
25	01-Feb-16	Lab-1	Linux-3	Lab-2	Li	
26	08-Feb-16	Lab-2	C-4	Lab-3	Li	
27	15-Feb-16	Lab-3	C-5	Lab-4	Li	
28	22-Feb-16	Lab-4	C-6	Lab-5	Li	
29	29-Feb-16	Lab-5	C-7	Lab-6	Li	
30	07-Mar-16	Lab-6	**C Revision**		Class Test	
31	14-Mar-16		**Student Choice**		Test Results	
32	21-Mar-16		*Practical Exam*			
33	28-Mar-16		Vacation			
34	04-Apr-16					
35	11-Apr-16					
	18-Apr-16		Exam Period Starts			

Lab-2 submission deadline Fri	05-Feb-16
Lab-3 submission deadline Fri	12-Feb-16
Lab-4 submission deadline Fri	19-Feb-16
Lab-5 submission deadline Fri	26-Feb-16
Lab-6 submission deadline Fri	04-Mar-16
	11-Mar-16

Lab-0: Orientation

Exam - Mandatory

Class Mock Exam
Thurs 10 March
2 to 3 pm

Results of Mock Exam

Practical Exam
Open Book
Mandatory
25% of your Final Grade
Tues 22 March
Two sessions TBA

Memory allocation

- We've seen how variables declared in function (or block) scope are "automatically" allocated memory, on the stack.
- This memory is also automatically deallocated when the function returns.
- There are limitations to this default mode of operation:
  - We can't make a variable whose memory lasts for longer than the function.
  - The Stack itself has size limits on memory use.

Scope refresher

This code prints…
the value 16

As g is in the file scope
the name is known inside f,
and is evaluated properly.

…but…
how is g *allocated*?

```
#include <stdio.h>

int g = 4;
void f(void);

int main(void) {
double myfloat;
f();
printf("%d\n",g);
}

void f(void) {
 g = g*4;
}
```

file scope
-
names "g" and "f" (and "main") are known throughout file

block scope (main function)
name "myfloat" known only in here
(we also know g as it's in file scope)

block scope (f)
we know g as in file scope

Static allocation

- Variables declared outside functions (in the "file scope") can't be automatically allocated, by definition.

- Instead, they are "baked into" the code itself - the memory for them is *always* allocated, until the program ends.

- We call this "static allocation".

```
int f = 5;

int main(void) { …
```

memory for `f` is put aside at the the point of compilation (in the "data segment"), and set to 5 at that point!

External linkage

- Because file scope, static, variables are baked into code, we can use the linker to "match up" names for them between different files. (Just like we match up functions.)

- To stop the compiler putting aside more than one chunk of memory for them, though, we need to use the "`extern`" keyword.

- This tells the compiler "this variable is already being stored by another file - I just want to refer to it here".

file1.c

```
int my_global = 5;
```

memory for `my_global` is put aside in file1.c, and set to `5` initially.

file1.h

```
extern int my_global;
```

memory for `my_global` is just a name in file1.h, linker will match it up with file1.o

Static variables in functions

- We can also request that a variable be statically allocated inside a function.
- For this, we need the `static` keyword.
- `static` variables still have the *scope* of the containing block… but the memory for them exists (is *allocated*) for the entire program.

```
int f(int increase){
 static int counter = 0;
 counter += increase;
 return counter;
 }
```

`counter` is initialised exactly *once* - at the start of the program itself (*not* when f runs).

Annoyingly, `static` means something very different when used for a variable in the file scope.

```
static int g;
```

prevents `g` being seen by other files and the linker!

Static allocation example

```c
#include <stdio.h>


int function(int add){
        static int counter = 0;

        counter += add;
        return counter;
}
```

counter is `static`, so is set to 0 at start of program

every time function is called, counter increases

```c
int main(int argc, char * argv[]){

        int c, result, numparsed;
        for(int i=1;i<argc;i++) {
                numparsed =sscanf(argv[i],"num=%d",&c);
                if (numparsed!=1){
                        printf("argument %s is not valid, skipping\n", argv[i]);
                        continue;
                }
                result = function(c);
                printf("The value of c is now: %d\n", result);
        }

        return 0;
}
```

loop through command line arguments, for each one which is like "num=X" where X can be interpreted as an int…

…add X to the counter

mandel.c returns!

```c
#include <stdio.h>
#include <complex.h>
#include <math.h>
#include <png.h>
#include <stdint.h>
#include "initpng.h"

#define DIM 500

uint8_t mandel_kernel(double complex c);

int main(void) {

        uint8_t array[DIM][DIM];
        double complex c;

        for(int i=0; i<DIM;i++){
                for (int j=0;j<DIM;j++){
                        //align on range -2..2 in x and y
                        c = (4*(double)i/DIM)-2 + ((4*(double)j/DIM)-2)*I;
                        array[i][j] = mandel_kernel(c);
                }
        }

        FILE *fp = fopen("mandel.png", "wb");

        png_structp png = initpng(fp,DIM);

        uint8_t * rows[DIM];
        for(int i=0;i<DIM;i++) {
                rows[i] = (uint8_t *)array[i];
        }

        png_write_image(png, rows);
        png_write_end(png, NULL);

        fclose(fp);

}
```

What happens if we make DIM a bit bigger? Say, 3000?

```c
uint8_t mandel_kernel(double complex c) {
        uint8_t i = 100;
        double complex z = 0 + 0*I;
        while(cabs(z)<2.0 && i>0){
                z = cpow(z,2) + c;
                i++;
        }
        return i;
}
```

```
gcc -Wall mandel.c initpng.o -lpng -lm
./a.out


Segmentation fault: 11
```

Dynamic allocation

- The stack for any process is a limited size. (Default, 8 megabytes).
- Really big things won't fit into it, causing code to crash when automatically allocated them.
- To make something really large, we need to explicitly ask the operating system to put aside memory for us.
- We also need to tell the operating system when we're finished with it, so it can take it back.

calloc

- We can request memory with `calloc`.
- We need to include `stdlib.h` to use it.
- If calloc succeeds, it returns a `void *` pointer to the start of the memory it got for us.
- This memory is allocated from a thing called "The Heap"
- If it *fails*, the pointer will be `NULL`.
- You should *always* test to see if `calloc` failed!

`void *` **means "this is just a location, with no assumption about type"**

calloc

- The arguments to calloc are "the size of a unit of memory" and "how many of them you'd like".
- (In this it resembles fwrite and fread somewhat.)
- If successful, calloc will fill the newly allocated chunk of memory with `0`s.

Here, we allocate sizeof(int)*1000000 = 4000000 bytes (~4MB) of memory.
Because of pointer/array semantics, this is like an array: `int bigarray[1000000] = {0};`

```
int * bigarray = calloc(1000000,sizeof(int));
if (bigarray==NULL)
   exit(1); //couldn't get memory for bigarray
```

free

- When you are finished with memory that you got via `calloc`, you need to tell the OS that you're finished with it with `free`.

- `free` *must* be called on a pointer to memory that was allocated with `calloc`.

- If you call it with something else, your code will crash. (`NULL` is also safe, but won't do anything.)

```
free(bigarray);
bigarray = NULL;
```

To guard against accidentally trying to use a pointer to memory we just gave back, set the pointer to NULL immediately!

Common issues

- Because memory allocated with `calloc` is not released until `free`d, it is important to keep track of all of your dynamically allocated memory, and free it when you don't need it. (It won't go away at the end of the function!)

- A "memory leak" is a programming error where a piece of code keeps on consuming more and more memory, as it forgets to release dynamically allocated memory.

- (Or even loses track of pointers to memory it allocated!)

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

int * make_big_array(int size){

        int * array = calloc(size,sizeof(int));
        if (NULL != array)
                return array;

        exit(20);
}

int main(int argc, char * argv[]){

        int size;

        if (argc != 2)
                exit(21);

    if (1!=sscanf(argv[1],"mem=%d",&size)) {
            printf("Enter argument as \"mem=X\" where X is an integer\n");
            exit(22);
     }

    int * array = make_big_array(size);

    array[size/2] = size*3;
    printf("Element %d of array set to %d\n", size/2, array[size/2]);

    free(array);
}
```

this function uses calloc to allocate space large enough for our array. Because the memory is dynamically allocated, it's fine to pass the pointer back from the function (as it will remain allocated until we explicitly free it)

get user request for "mem=X"

allocate array of ints of that size (pointer/ array notation equivalence is useful!)

and just to prove we did it, set a value in the middle of the array

be sure to free array when done!

mandel.c (dynamic)

```c
#include <stdlib.h>

#define DIM 3000

uint8_t mandel_kernel(doubl

int main(void) {

        uint8_t (*array)[DIM] = calloc(DIM,sizeof(uint8_t[DIM]));
        if (array == NULL)
                exit(1);

        double complex c;

        for(int i=0; i<DIM;i++){
                for (int j=0;j<DIM;j++){
                        //align on range -2..2 in x and y
                        c = (4*(double)i/DIM)-2 + ((4*(double)j/DIM)-2)*I;
                        array[i][j] = mandel_kernel(c);
                }
        }

        FILE *fp = fopen("mandel.png", "wb");

        png_structp png = initpng(fp,DIM);

        uint8_t * rows[DIM];
        for(int i=0;i<DIM;i++) {
                rows[i] = (uint8_t *)array[i];
        }

        png_write_image(png, rows);
        png_write_end(png, NULL);

        fclose(fp);
        free(array);
}
```

need stdlib.h for prototypes for calloc, free

mixed array/ptr notation needed here - calloc gives us a pointer, which we need to interpret as a pointer to DIM sized arrays (C's array model lets us use array as "normal" for the rest of the code.)

remember to free array when we're done!

Random Numbers

- All the code we've written is deterministic - it does the same thing every time.
- Some things we might want to simulate are apparently not so - nuclear decay, "noise" on signals, ages of a person you bump into.
- Sometimes we also just want to generate "different values", to explore the phase space of a problem.
- So, how do we generate "random variation"?

Pseudo Random Numbers

- Most of the time, we don't really care about having truly "random" sequences.

- What we want are sequences which have statistical properties of a random sequence:
    - each value is hard to predict from the ones before it (**uncorrelated**)
    - the probability of any value is the same (**uniform**)

- We can imagine a *pseudorandom* number generator, which produces such a sequence, but deterministically, from some kind of internal state.

A (bad) example PRNG

0**54**1

$\mathbf{54}^2 = 2916$

2**91**6

$\mathbf{91}^2 = 8281$

8**28**1

$\mathbf{28}^2 = 0784$

0**78**4

$\mathbf{78}^2 = 6084$

6**08**4

$\mathbf{08}^2 = 0064$

0**06**4

$\mathbf{06}^2 = \ldots$

…

- John Von Neumann wrote this PRNG in 1949.
- We start with a "seed" value: here 0541.
- Each value in the sequence is the square of the middle digits of the value before it.

PRNGs - flaws

0**00**0	0**00**1	0**00**2	9**00**8		1**45**3		1**24**3
0**00**0	0**00**0	0**00**0	0**00**0		2**02**5		0**57**6
0**00**0	0**00**0	0**00**0	0**00**0		0**00**4		3**24**9
0**00**0	0**00**0	0**00**0	0**00**0	...	0**00**0	...	0**57**6
0**00**0	0**00**0	0**00**0	0**00**0		0**00**0		3**24**9
0**00**0	0**00**0	0**00**0	0**00**0		0**00**0		0**57**6
0**00**0	0**00**0	0**00**0	0**00**0		0**00**0		3**24**9
0**00**0	0**00**0	0**00**0	0**00**0		0**00**0		0**57**6
...	...	...	...		...		...

- Badly designed PRNGs can have unexpected behaviour for some seed values.

- Here, starting with any sequence containing 00, 45, 24 (or 57) generates a very predictable ("non-random") sequence!

PRNGs - flaws

- Even much more sophisticated PRNGs, which don't have obvious flaws, can have more subtle issues.

- For example: IBM's **RANDU** PRNG, which was widely used in the 1960s.

- Although the output "looked random", it actually had some extremely strong correlations, which made the output untrustworthy.

- Many millions of hours of simulations had to be redone once this was discovered!

PRNGs - advantages

- Because a PRNG generates a sequence from an internal state, which is set by a seed value…
- …the same seed will always give the same sequence.
- So, we can directly compare the result of changes to code, by running it with the same seed as before.
- (This is also the mechanism used for "seeded" or "daily" runs in various computer games and entertainments - Minecraft, Spelunky, Nuclear Throne, XCOM(2) etc)

PRNGs - advantages

- The alternative to PRNGs is to try to "collect" randomness from the real world.
- "Hardware Random Number Generators" exist, but must gather truly random sources - radioactive decay, or shot noise in electronic circuits - which limits the rate at which they can work.
- PRNGs are usually faster than HRNGs; and you can always rely on them being available!

PRNGs in C

- The C Standard Library provides some functions for generation of pseudorandom sequences.
- These functions are prototyped in `stdlib.h`, so you need to `#include` it to use them.
- `void srand(unsigned int seed)`
  - set seed for PRNG to provided seed value (default is `0`)
- `unsigned int rand(void)`
  - get next value from PRNG (integer in range `0` to `RAND_MAX`)

Example - continuous uniform variates

```
double zero_to_one(void) {
 return rand()/ (double) RAND_MAX;
}
```

`rand()` will only generate values from
0 to **RAND_MAX**
If we re-scale those values by dividing by **RAND_MAX**,
then we will clearly get a number between **0** and **1**.
(Remember to cast to `double`, to avoid integer division)

Example - discrete uniform variates

```
unsigned int d6(void) {
 return 1 + floor(rand()*6.0/RAND_MAX);
}
```

`rand()` will only generate values from **0** to **RAND_MAX**

If we re-scale those values by dividing by **RAND_MAX**, then we will clearly get a number between **0** and **1**.

We can then multiply by the actual range of integer values we want (here **6**), and use the `floor()` function to turn the floating point values back into integers.

`floor()` will give us values in range **0-5** here (it rounds down), so add **1** to give the numbers on a die.

Example - seeding rand from system time.

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

int main(void) {
 /* time(TIME) returns specified TIME as seconds
  since the "Unix Epoch" (midnight 1 Jan 1970).
 If you pass it NULL, it returns the current time. */

 srand(time(NULL));

 //print 100 pseudo random numbers in sequence
 for(int i=0; i<100; i++) {
  printf("%d\n", rand() );
 }

 return 0;
}
```

Limitations of rand()

- `rand()` is convenient, being in the C Standard Library, but it has limitations.

- As `srand()` takes an unsigned int as a seed, there are only `UINT_MAX` possible sequences that rand can produce (one per seed).

- The C Standard doesn't require `rand` to have a particularly long sequence length (for portability). In some versions of the library, `rand` can have quite short repeat lengths, or fail some statistical tests.

Beyond rand()…

- For serious work requiring pseudorandom sequences with good statistical properties, you should use a dedicated PRNG from a library designed for this.

- Mersenne Twister and WELL, for example, are two widely used and very good PRNGs which you should consider.

- (For cryptography, even these are not sufficient - consider special cryptographically strong PRNGs, or actually taking random numbers from outside machine.)

References

- Mersenne Twister
- http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
- WELL
- http://www.iro.umontreal.ca/~panneton/WELLRNG.html
- Example use for PRNGs:
- http://moodle2.gla.ac.uk/mod/resource/view.php?id=149996

C Program Structure

Programs Contains Data and Instructions

```
#include <stdio.h>

int main()
{
    int i, j;
    i = 3;
    j = i * 2;
    printf("%d doubled is %d\n",i, j);
    return 0;
}
```

Starts with one or more standard lines which tells the linker where to find information.

Program consists of one (or more) functions.

Function composed in terms of:
DECLARATIONS and STATEMENTS
(which end with a semi-colon!)

" j = i * 2;" is an expression statement, composed of:
Variables (i and j);
*Operators (= and *);*
Literal (2).

"printf" is a function-call statement.

Return code is either '0' (success) or non-zero (error-code – failure); can be used by calling script.

1

Compiling a Program

Under Linux command-line compilers are most common (as oppose to the interfaces you get, say, on Windows machines).

The following command compiles a program contained in a file called myProg.c

```
> gcc  -o myProg  myProg.c
```

which tells the compiler to read, compile, and link the source file myProg.c and write the executable code into the output file myProg

The program is then run on a linux machine using the command:

```
> ./myProg
```

Where the ./ tells it to look in your 'present working directory' (in case this is not in your 'path' – refer to Linux lectures).

2

Outline

Literal values

* A literal is an explicit value written into the C source code itself.

`99`	Integer type
`'a'`	Character type
`5.67`	Floating point number - float type

* All values in C, including literals, have a *type* that defines what kind of value they are.

Types

* The type of a value determines how the computer interprets the pattern of bits that represent it in memory.
* Types also determine how many bits are needed to store the value in memory.
* C has several built-in types (which mostly represent ways in which the CPU itself can deal with data).
* C also allows you to define "composite" types derived from those built-in types.

Computer Memory

➢The smallest bit of information stored by a computer is a 1 or 0 - this is called ***a bit*** and, therefore, the natural number systems for computers, at the lowest level, is binary.

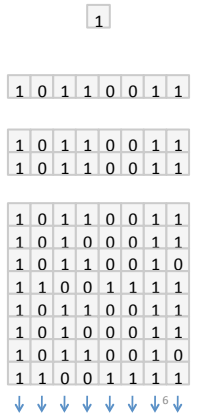
➢Eight *bits* are called one ***byte***.

➢A complete unit of information is called a ***word*** which consists of one or more bytes.

➢Thus, you can visualise memory as an array of memory-cells that has eight columns (or perhaps better visualised as n times eight columns, where n is the number of bytes in a word) and a large number of rows: Thus, a 64 Mbyte memory could have 8 columns and 64 million rows*

➢The location of data in memory is called its ***address***

*Well not really; 1KB = 1024 bytes (2**10)



Integer Types

* Representations of integer values.
* Several different types, for different memory use (and consequent max and min representable values).
* Each type also has a signed and an unsigned (positive only) subtype (default to signed).

Types on Intel 64bit	`short` (`unsigned short`)	`int` (`unsigned int`)	`long` (`unsigned long`)
Max value	32767 (65535)	2,147,483,647 (4294967296)	2^{63}-1 (2^{64}-1)
Min value	-32768 (0)	-2,147,483,648 (0)	-2^{63} (0)
Size (bytes)	2	4	8

E.g. unsigned int i = 321;

7

Floating-Point Types

* Limited precision representations of real numbers.
* http://moodle2.gla.ac.uk/mod/page/view.php?id=149958

Types on Intel 64bit	`float`	`double`	`long double`
Max value	$\pm 3.4 \times 10^{38}$	$\pm 2.2 \times 10^{308}$	$\pm 1.18 \times 10^{4932}$
Smallest non-zero value	$\pm 1.18 \times 10^{-38}$	$\pm 2.2 \times 10^{-308}$	$\pm 3.65 \times 10^{-4951}$
Decimal precision	7 sig. fig.	16 sig. fig.	19 sig. fig.
Size (bytes)	4	8	16 (often)

Other Types

* For non-numeric data, there are a small set of other types.
* `chars` are single characters of text (the symbol '9' is distinct from the number 9). Internally they are represented as *numbers*, indices into a table of symbols.
* This means that we can 'add' values to `chars`, to get the value that many spaces along in the table (`'a' + 1` is `'b'`).

Types on Intel 64bit	`void`	`char`
Kind of data	"No type": used to indicate explicitly that there is no type or value expected.	Single characters (`'a'`, `'G'`, `'9'`, `':'`)
Size (bytes)	N/A (effectively 1)	1 (by standard)

ASCII: American Standard Code for Information Interchange.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

10

Other ASCII Characters

* We can include characters which are not printable (or, like the character `'`, can't be written in as a literal without terminating the quotes) using the \ symbol to 'escape' them.
* So, `'\''` is the `'` as a `char`, `'\n'` is the result of pressing enter, `'\t'` is a tab space and `'\\'` is an actual \ as a character.
* Some more special types will be covered later in the course.

Operators

* Operators are built-in methods for manipulating values.
* C provides a small set of operators for arithmetic, logical and utility purposes, most of which translate directly to operations that the CPU itself has built-in.
* We will spend some slides listing the most relevant ones to this course (there are a few others that you can look up, and some we will introduce later on).
* Just as with mathematical operators, operators in C have precedence rules to resolve compound expressions like `2*3+4` . We can always put parentheses around a subexpression that we want to force to be calculated first, to override precedence: `2*(3+4)`.

Arithmetic Operators

* The Arithmetic Operators work on any data type that is numeric.
* They are defined to work within the type of the values they are given - division of two integer type value produces the nearest *integer* value as the result!
* Mixing two types in an operation (e.g. `1.2 * 15`) performs the calculation as if both values had the "widest" type in the operation (that is, the type that can store the largest values). Usually, this means that you'll get a floating-point calculation if you mix floats with ints.

Operation	Symbol	Precedence	Example
Addition	+	Low	2+4 (=6)
Subtraction	-	Low	2-4 (= -2 if signed type)
Multiplication	*	High	2*4 (=8)
Division	/	High	2/4 (=0 if integer type!)
Remainder	%	High	2%4 (=2)

Relational Operators

* Relational operators compare two values, and return a true (1) or false (0) result dependant on if the given relation is the case.
* In general, "non-zero" is taken to mean "true" for the purpose of assigning truth or falsity to integers.

Operation	Symbol	Precedence	Example
Equality	==	Low	2==4 (`false`)
Greater Than	>	Low	2>4 (`false`)
Less Than	<	Low	2<4 (`true`)
Greater than or equal	>=	Low	2>=4 (`false`)
Less than or equal	<=	Low	2<=4 (`true`)
Not equal	!=	Low	2!=4 (`true`)

E.g. if (x == 4) "do something"

Logical & Misc. Operators

* Logical operators combine two true or false values using Boolean logic.
* Note that, as all non-zero integer values are considered `true`, these operators can be used on integer types as well as the logical values themselves.
* The `sizeof` operator returns the amount of bytes used to represent the value in memory. Often used in code which needs to run on many different types of machine (where the size of, say, `int` might be different).

Operation	Symbol	Precedence	Example
Logical AND	&&	Low	`1 && 0` (0)
Logical OR	||	Low	`1||0` (1)
Logical NOT	!	Low	`!1` (0)
Number of bytes to represent value	`sizeof()`	Highest	`sizeof`(321) (4)

Casts: Converting types

* As mentioned above, some operators automatically change the type of their operands so that they all match in type.
* We can also explicitly change the type of a value using a `cast`.
* Specifying the name of a type surrounded by ( ) tells the compiler to convert the following value to the indicated type.
* For example, we can cast a `float` to a `double` like so:
  myDouble = (float) myfloat;
* Obviously, casting a value to a type which does not support it (casting to int a value bigger than the largest value an int can store, for example) will cause a loss of information or unexpected effects.

Variables

* So far, we can perform calculations, but we have nowhere to store their results.
* *Variables* provide a way to store values of a given type, with an associated identifier ("variable name") to refer to the storage location.
* Variable names may be of any length and can be made up of any combination of letter, digit and _ characters which do not look like a literal value or a C *keyword*. Can not start with a number
* Because the identifier refers to the *storage location*, we can change the value stored in a given variable with no inconsistency.
* Variables must be declared with a type, which defines how values stored in them will be interpreted.

Variables

* So far, we can perform calculations, but we have nowhere to store their results.
* *Variables* provide a way to store values of a given type, with an associated identifier ("variable name") to refer to the storage location.
* Variable names may be of any length and can be made up of any combination of letter, digit and _ characters which do not look like a literal value or a C *keyword*. Can not start with a number
* Because the identifier refers to the *storage location*, we can change the value stored in a given variable with no inconsistency.
* Variables must be declared with a type, which defines how values stored in them will be interpreted.

Declaring Variables

* Before you can use a variable identifier, you must *declare* it, letting the compiler know that you are going to want to use a name to refer to a location for storing data of a given type.
* Variable declarations are of the form type of data we want to store names we want to use for the storage locations, where we separate names by commas and end with a ;.

```
int a ;
double b, c;
unsigned long d = 2;
char e = 'a';
double g = d * 2.0;
```

Declaring Variables

* If you use the simple declaration form, then C does not guarantee what value a variable will contain to start with.
* To ensure that your variables start with a known contents, you can optionally assign an initial value to them.
* Valid values are literals, or the values of any variable you've previously declared, or the results of calculations involving them.

```
int a ;
double b, c;
unsigned long d = 2;
char e = 'a';
double g = d * 2.0;
```

Operators on Variables.

* All of the previously mentioned operators will work on the values stored in variables (with appropriate types).
* In addition, there are some operators specifically designed to modify the values stored in variables.

Operation	Symbol	Precedence	Example
Assignment	=	Low	a = 2; (a contains value 2)
Add to stored value	+=	Low	a += 2 (a == 4)
↑ Equivalent versions exist for the other arithmetic operators. ↑			
Increase / Decrease by 1	++/--	High	a++ (a == 5)

a += 2; is equivalent to a = a + 2; += -= *= /=

Gotchas…

E.g. if (x == 4) "do something"
This uses the equality operator == to test if variable x is equal to the value 4, or not.

On the other hand, writing:
if (x = 4) "do something"
This (incorrect statement) uses the assignment operator = to set the variable x to 4. The if-test will always evaluate "true" (x is "non zero") because the value of x has been set to four!

a++ increments a by one; ++a does the same; but we will see that they behave differently in some circumstances:

Eg. if(a++ == 3) is not necessarily the same as if (++a == 3). The first performs the operation first and then increments 'a'; the latter does the increment first. So if a=2

5

C – Lecture 2

- Printing to the screen – printf()
- Flow Control
  - Conditional Branching
    - The `if` statement.
    - The `switch`,`case` construct.
  - Loops
    - The `do… while` construct.
    - The `for` loop.
    - Breaking out of loops - `continue` and `break`
- Scope

The printf() Function

A standard C function included in the library <stdio.h>-used to print to the screen.

Standard form is: printf(format, expression-1, expression-2, ...)

Example: printf("Twice %d is %d\n",i, j);

"Format" "Expressions"

```
#include <stdio.h>

    int main()
    {
        int i, j;
        i = 3;
        j = i * 2;
        printf("Twice %d is %d\n",i, j);
        return 0;
    }
```

The %d is called the "conversion" and the "d" means this is an integer-conversion which tells the compiler to interpret the arguments (i and j) as integers. For floating point numbers, %f is used.

Note: You have to use the conversion that correctly matches your expression.

Definition contains "expression-n" (not "variable-n") because could replaced j with i * 2

7

printf Format Statements

```
printf ("Twice %d is %d \n", term, 2*term);
```

Besides %d there are other conversion specifications, here is an overview of the most important ones:

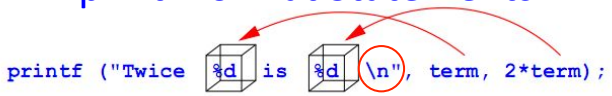
Conversion	Argument Type	Printed as
%d	integer	decimal number
%f	float	[-]m.dddddd (details below)
%X	integer	hex. number using A..F for 10..15
%c	char	single character
%s	char *	print characters from string until '\0'
%e	float	float in exp. form [-]m.dddddde±xx

In addition, the precision and additional spaces can be specified:

%6d	decimal integer, at least 6 characters wide
%8.2f	float, at least 8 characters wide, two decimal digits
%.10s	first 10 characters of a string

8

printf Format Statements



Special characters or escape characters starting with '\' move the cursor or represent otherwise reserved characters.

Character	Name	Meaning
\b	backspace	move cursor one character to the left
\f	form feed	go to top of new page
\n	newline	go to the next line
\r	return	go to beginning of current line
\a	audible alert	'beep'
\t	tab	advance to next tab stop
\'	apostrophe	character '
\"	double quote	character "
\\	backslash	character \
\nnn		character number nnn (octal)

Conditional Branching

* So far, there has been no opportunity for decision making in the code written.
* It would be useful to be able to choose to do one thing, or another thing, on the basis of the result of some test.
* This concept is called a "conditional branch", from the metaphor of a tree which splits into multiple branches as it grows.

The if statement.

* The most basic conditional structure in C is the `if/if else/else` construct.
* This block is executed only if the test evaluates to `true`.

```
if (test1) {
  some statements;
}
```

Example

```
/*************************************************
* if.c - Program to illustrate "if"        *
*                                                *
* David Britton, Jan 2014                        *
*                                                *
*************************************************/

#include <stdio.h>

        int main()
        {

        int x = 4;
        if (x == 3)  {
          printf("x = %d \n", x );
        }
        else if (x > 3) {
          puts("x is greater than 3");
        }
        else
          puts("x is less than 3");

      return 0;
      }
```

The switch,case construct.

* We can use long chains of `else ifs` to test a variable for many different values, and do different things dependant on them.
* However, this is ugly and inefficient, and also doesn't let us reuse common code between some of the blocks.
* The `switch,case` construct is a more natural fit for this kind of problem.

switch, case

* `switch` is given an expression which will evaluate to an integer value.
* Each case has an integer value which will be compared to the switch expression, followed by a `:`.
* Code in the `switch` block is executed starting *immediately after* the case label which matches value.
* If no case matches the value, execution starts after the (optional) `default:` label.

```
switch (expression) {
  case value1:
    some statements;
    break;
  case value2:
    more statements;
  default:
    further statements;
}
```

switch, case

* If the expression were equal to *value1*, these statements would be executed.
* The `break` statement stops execution when encountered, and moves to the end of the block (outside the switch itself).
* If the expression instead were equal to *value2*, these statements would be executed.
* The lack of a `break` allows all the statements after the case label to be executed, including those after subsequent labels.

```
switch (expression) {
  case value1:
    some statements;
    break;
  case value2:
    more statements;
  default:
    further statements;
}
```

Example

```
/************************************************
* switch.c – Program to illustrate "switch"     *
*                                               *
* David Britton, Jan 2014                       *
*                                               *
************************************************/

#include <stdio.h>

        int main()
        {

        int x = 4;
        switch (x) {
            case 1:
                puts("This is case-1");
                break;
            case 2:
                puts("this is case-2 with drop-through");
            case 3:
                puts("this is case-3 or drop-through from case-2");
                break;
            default:
                puts("this is the default case");
        }
      return 0;
      }
```

Loops

* Now we can do different things based on a test, but we can only do each thing *once*.
* There are lots of processes that essentially reduce to either:
  Do (operation) on (every member) of (some list of items).
  Do (operation) until (some condition is met).
* Both are examples of loop structures, which repeat a core operation many times.
* C provides several loop constructs which are designed to handle the first or second cases particularly naturally.

The do... while construct.

* The `do ... while` construct repeats until a condition is met.
* Every iteration:
  1. The statements in this block are executed.
  2. The test is performed.
  3. If `true`, we jump back up to the start of the block.
  4. If `false`, we continue on to the next statements after the block.

```
do {
  some statements;
} while (test) ;
```

While Construct

```
while (test) {
    ...
    some statements;
    ...
}
```

The for loop.

* Whilst a do-loop repeats until a condition is met, a `for`-loop is intended for situations where you are explicitly counting loops (or doing the same thing for multiple items).
* For example, multiplying all the elements of a list of values by a number, or printing something out a certain number of times.

```
for(init ; test ; iter  )
{
  some statements;
}
```

The for loop.

* The `for` statement contains three semicolon separated expressions.
* The first (and only the first) time we encounter the statement, the init expression is executed. Usually this is used to set a starting value for a "counter" variable.
* In C99 and later, we can declare a variable and initialise it in the init expression, limiting the scope of our counter to the loop itself. (i.e. `int i = 0`)

```
for(init ; test ; iter  )
{
  some statements;
}
```

The for loop.

* Every time (including just after the init), the test is performed.
  - If `true`, we execute this block.
  - If `false`, we skip to the end of block and continue on.

```
for(init ; test ; iter  )
{
  some statements;
}
```

The for loop.

:

* If we did execute the block, we then execute the iter expression.
* Usually this is used to update a counter variable (i.e `i++`), or get a new bit of data to work on.
* We then jump up to the top of the block again, and do the test...

```
for(init ; test ; iter  )
{
  some statements;
}
```

Example

```
/*********************************************
* static.c – Program to illustrate static      *
*                                              *
* David Britton, Jan 2014                      *
*                                              *
*********************************************/

#include <stdio.h>

        int main()
        {

        int temporary = 3;
        for (int i = 0; i < 4; i++) {
          int temporary = 0;
          static int permanent = 0;

          temporary = temporary + 1;
          permanent = permanent + 1;

          printf("temporary and permanent = %d %d \n",temporary, permanent);
        }
//        printf("temporary and permanent = %d %d \n",temporary, permanent);
       printf("temporary = %d \n",temporary);

      return 0;
      }
```

Breaking out of loops - `continue` and `break`

* While the loop structures above are usually sufficient as they are, sometimes we might need to modify their flow a little bit for special cases.
* The `continue` and `break` statements let alter the flow of execution inside the loop block.
* We've already met `break` when we covered `switch`.
* It works similarly in a loop block, jumping out of the loop block immediately, and continuing with whatever statements happen after it.
* The `continue` statement is like a `break`, but just for that iteration of the loop.
* We jump to the end of the loop block, but still perform the *test* (and, in a `for` loop, the *iter* expression) to see if we should keep on looping.

Scope

* A variable name is not always defined over the entire length of a program.
* The part of the program in which a variable name is defined is called its *scope*.
* Variable names declared outside of any block have *file scope*.
* Variable names declared inside a block have *block scope*.

File Scope

* The variable `a` has file scope.
* It is defined outside any blocks, and so, by default the name `a` will refer to it everywhere in the file.

```
int a = 5;

int main(void) {
  /* Some statements */
  {
    int b = 6;
  }
}
```

Block Scope

* The variable `b` has block scope.
* It is defined inside a block, and therefore the name `b` will only refer to it for statements inside that block.

```
int a = 5;

int main(void) {
  /* Some statements */
  {
    int b = 6;
  }
}
```

Block Scope

* Block scope variables can *shadow* variables of the same name in an wider scope.
* The variable a in the block scope shadows the file scope a inside that block.
* Any calculation using the name a in that block will use the block scope a.

Shadowing is generally regarded as bad practice!

```
int a = 5;

int main(void) {
  /* Some statements */
  {
    int a = 77;
    int b = a;
    // b == 77, not 5!
  }
  //returns the value 5
  //block a out of scope.
  return a;
}
```

Allocation (of memory)

* Just as variable *names* have scope over which they have meaning, the memory used to store their values also has a limited duration, or *lifetime*.
* Block scope variables usually backed by *automatic* allocation. The storage used for the variable is allocated to it when the block it is declared in starts, and then automatically given back when we encounter the end of the block.
* File scope variables, and block scope variables declared with the `static` keyword before their type, have *static* allocation. While the name associated with the storage may only have a limited scope, the storage itself is kept for the duration of the program, along with any values in it.

So that you can re use value
Next time enter block

Example

```
/*************************************************
* static.c - Program to illustrate static       *
*                                               *
* David Britton, Jan 2014                       *
*                                               *
*************************************************/

#include <stdio.h>

        int main()
        {

        int temporary = 3;
        for (int i = 0; i < 4; i++) {
          int temporary = 0;
          static int permanent = 0;

          temporary = temporary + 1;
          permanent = permanent + 1;

          printf("temporary and permanent = %d %d \n",temporary, permanent);
        }
//        printf("temporary and permanent = %d %d \n",temporary, permanent);
       printf("temporary = %d \n",temporary);

      return 0;
      }
```