P2T 2017
C Lecture 8

Dr Gordon Stewart
Room 427, Kelvin Building
gordon.stewart@glasgow.ac.uk

Source material: Prof David Britton / Dr Sam Skipsey

Pseudorandom Numbers

Random Numbers

- Your code so far has all been **deterministic**: it does the same thing every time it is run

- Sometimes we want to produce a statistical output by sampling many different possible events…

- …or model events with random aspects:

  - shuffling a pack of cards

  - radioactive decay

  - signal noise

Random and Pseudorandom

- Often we don’t care about true randomness
- We want a sequence with the statistical properties of a random sequence:
  - **Uncorrelated:** each value is hard to predict from the ones before it
  - **Uniform:** the probability of each possible value occurring is the same
- A **Pseudorandom** Number Generator uses an algorithm to produce such a sequence deterministically

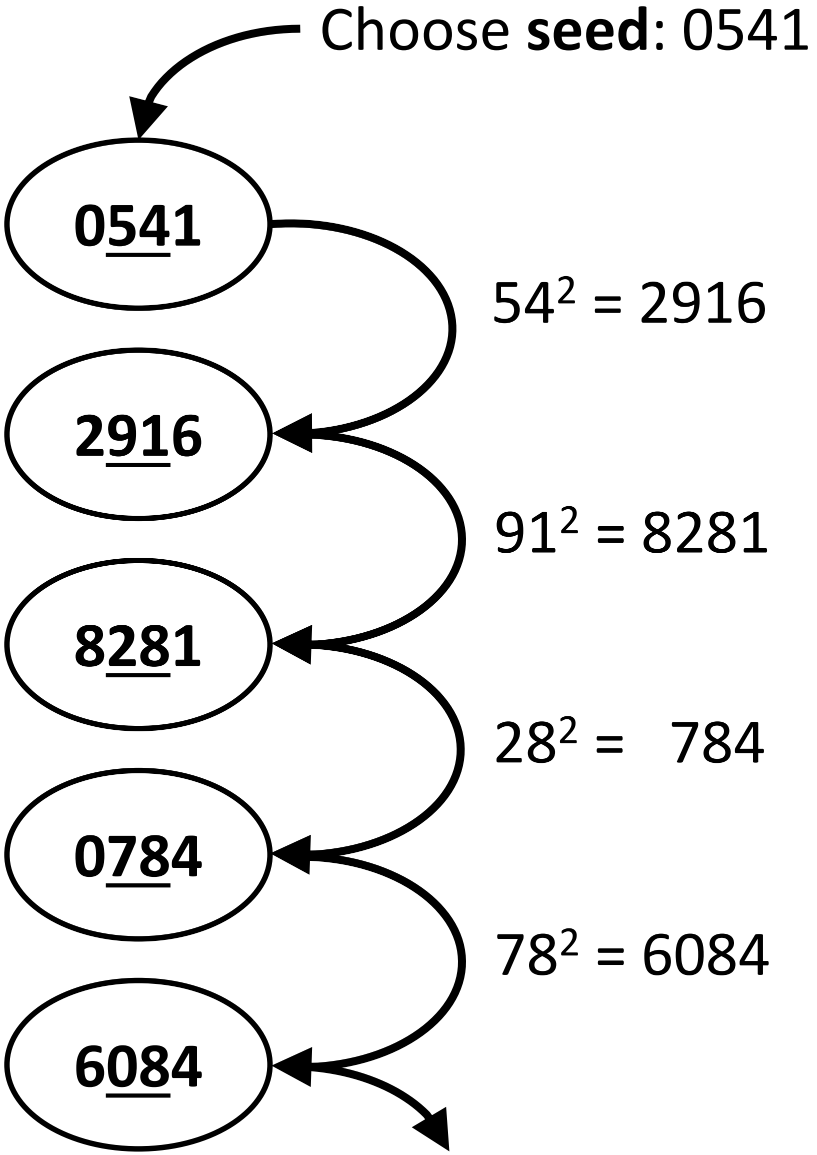
Pseudorandom Number Generators

- PRNGs use internal **state** to generate a sequence of pseudorandom numbers

- For simple PRNGs, this state may just be the previous random value

- John von Neumann invented the **middle-square method**

- This generates successive values by taking the square of the middle two digits of the previous value



Choose **seed**: 0541

0541

$54^2 = 2916$

2916

$91^2 = 8281$

8281

$28^2 = 784$

0784

$78^2 = 6084$

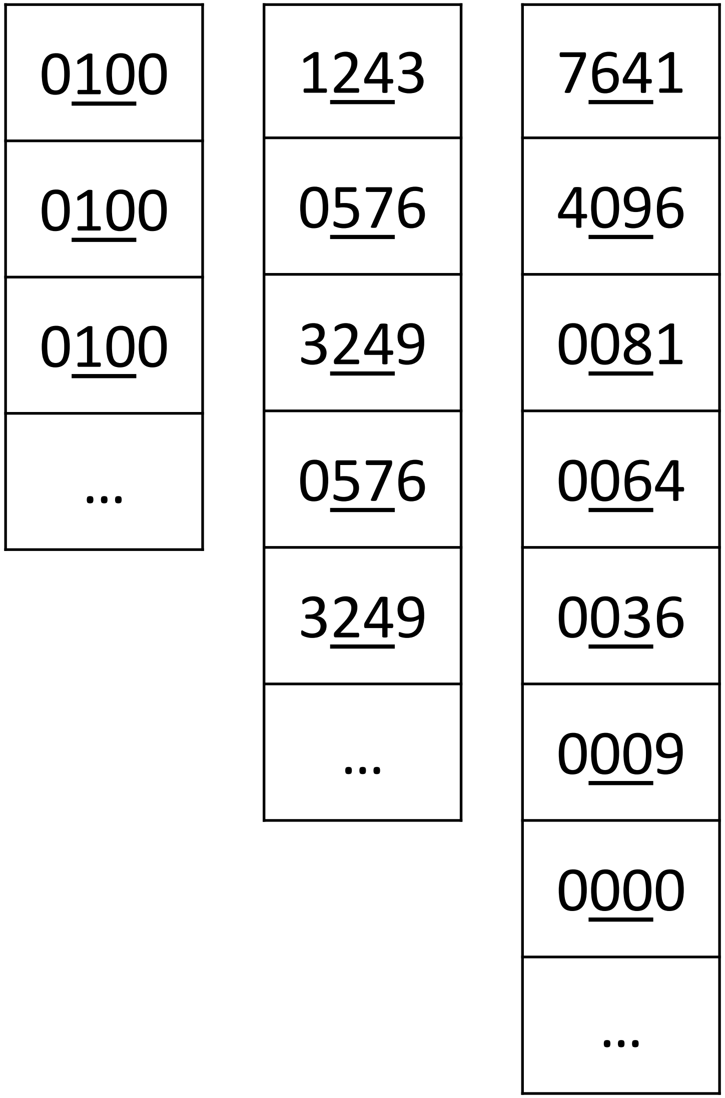
6084

PRNGs – Disadvantages

- PRNGs use internal state to generate next value, so if internal state repeats the sequence does too

- Maximum time before repetition is the number of possible values of the internal state, but certain sequences can repeat much more quickly

- PRNGs can also have subtle problems, e.g. IBM's **RANDU**



$0\underline{10}0$
$0\underline{10}0$
$0\underline{10}0$
…

$1\underline{24}3$
$0\underline{57}6$
$3\underline{24}9$
$0\underline{57}6$
$3\underline{24}9$
…

$7\underline{64}1$
$4\underline{09}6$
$0\underline{08}1$
$0\underline{06}4$
$0\underline{03}6$
$0\underline{00}9$
$0\underline{00}0$
…

PRNGs – Advantages

- Using the same **seed** always produces the same sequence:
  - Can change other parts of the code and check results are consistent, or compare results from different computers
- PRNGs can be much faster than collecting true random values
- Hardware Random Number Generators exist but are rare – don't expect to find one in your PC!
  - They sample random signals: thermal noise, nuclear decay, counting photons…
  - Performance limited by need to measure real effects

PRNGs in C

- The C Standard Library includes some functions to generate pseudorandom sequences in **`stdlib.h`**:

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

- **`srand(value)`**: Set seed for PRNG to **`value`** (takes an **`unsigned int`**, default is **`1`**):

  ```
  srand(123); // Set seed to 123
  ```

- **`rand()`**: Get next random value (range **`0`** to **`RAND_MAX`**):

  ```
  unsigned int x = rand();
  ```

Example: Continuous Uniform Variates

- Aim is to generate a floating-point value in a certain range
- **`rand()`** returns an integer between 0 and **`RAND_MAX`**
- Re-scale values by dividing by **`RAND_MAX`** (cast to **`double`** to avoid integer division)

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

Example: Seeding with System Time

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

int main(void)
{
    /* time(TIME) returns given TIME as seconds
       since UNIX epoch (midnight 1 Jan 1970).
       time(NULL) returns the current time. */
    srand(time(NULL));

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

    return 0;
}
```

Limitations of `rand()`

- Convenient, but has limitations compared to state-of-the-art PRNGs

- Seed is **`unsigned int`**, so **`rand()`** can produce at most **`UINT_MAX`** different sequences (one per seed)

- C standard defines **`rand()`** to be **portable** so range can be surprisingly small

- **`rand()`** is fine for testing and when you don't need high quality statistics, but not for serious research

Beyond `rand()`

- Many external libraries provide high-quality PRNGs with good statistical properties:

  - Mersenne Twister
    http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

  - WELL
    http://www.iro.umontreal.ca/~panneton/WELLRNG.html

- PRNG example use case:

  http://moodle2.gla.ac.uk/mod/resource/view.php?id=149996

C Revision

Types

- Every value in C has a **type**
- The type of a value determines how the computer interprets the bit pattern that is stored in memory

	`int`	`long`	`float`	`double`	`char`	**(pointer)**
Holds	integer	integer	floating-point	floating-point	ASCII character	address of memory location
Unsigned Version?	`unsigned int`	`unsigned long`			`unsigned char`	
Format String	`%d`	`%ld`	`%f`	`%lf`	`%c`	`%p`

Variables

- Variables are used to store values

- A variable **declaration** does two things:

  1. Allocates memory for a value of a certain type
  2. Assigns a **variable name** to that value so you can refer to it

$$\underbrace{\texttt{unsigned int}}_{\textbf{type}}\ \underbrace{\texttt{numberOfStudents}}_{\textbf{variable name}}\texttt{;}$$

- You can assign an initial value at the same time:

```
int numberOfStudents = 80;
char operator = '+';
```

Statements and Blocks

- A **statement** is a complete instruction
- Statements end with a semicolon
- A **block** is a set of statements surrounded by **{ }**:

```
{
   veloSquared = velocity * velocity;
   energy = 0.5 * mass * veloSquared;
}
```

- Most structures (e.g. loops) use blocks to group the code that they apply to

Scope and Allocation

- The **scope** of a variable is the part of the program in which its name is defined

- Variables defined outside any block have **file scope** and are in scope throughout the file after their definition

- Variables defined within a block have **block scope**, and go out of scope when the block ends

```
int x = 12;

int main(void) {
   int y = 3;
   int z = x * y;

   while (z >= 0) {
      int n = 1;
      z -= n++;
   }

   printf("z = %d", z);

   return 0;
}
```

Scope and Allocation

- The **scope** of a variable is the part of the program in which its name is defined

- Variables defined outside any block have **file scope** and are in scope throughout the file after their definition

- Variables defined within a block have **block scope**, and go out of scope when the block ends

```
int x = 12;

int main(void) {
  int y = 3;
  int z = x * y;

  while (z >= 0) {
    int n = 1;
    z -= n++;
  }

  printf("z = %d", z);

  return 0;
}
```

Scope and Allocation

- The **scope** of a variable is the part of the program in which its name is defined

- Variables defined outside any block have **file scope** and are in scope throughout the file after their definition

- Variables defined within a block have **block scope**, and go out of scope when the block ends

```
int x = 12;

int main(void) {
   int y = 3;
   int z = x * y;

   while (z >= 0) {
      int n = 1;
      z -= n++;
   }

   printf("z = %d", z);

   return 0;
}
```

Scope and Allocation

- The **scope** of a variable is the part of the program in which its name is defined
- Variables defined outside any block have **file scope** and are in scope throughout the file after their definition
- Variables defined within a block have **block scope**, and go out of scope when the block ends

```
int x = 12;

int main(void) {
    int y = 3;
    int z = x * y;

    while (z >= 0) {
        int n = 1;
        z -= n++;
    }

    printf("z = %d", z);

    return 0;
}
```

Scope and Allocation

- **Allocation** concerns the lifetime of the memory used to store a value
- The allocation range is the period, while the program is running, during which memory is allocated to store something
- Automatically allocated variables have memory allocated when the block in which they are contained starts, which is then handed back when the block ends

Simple C Program

```
#include <stdio.h>

int main(void)
{
    // Don't forget to include comments!
    printf("Hello, World!\n");
    return 0;
}
```

- Must **compile** the code to generate an **executable**:

```
gcc -Wall -std=c99 hello.c -o hello
```

-Wall: turn on warnings

-std=c99: enable C99 support

hello.c: code

hello: executable name

Flow Control

- **Conditionals** are used to branch within your code:
  - `if`
  - `switch`
- **Loops** are used to repeat blocks of code:
  - `for`
  - `while`

Conditionals

```
if (x == 1)
{
    // Do thing A
}
else if (x > 1 && x <= 4)
{
    // Do thing B
}
else
{
    // Do thing C
}
```

```
switch (x)
{
    case 1:
        // Do thing A
        break;
    case 2:
    case 3:
    case 4:
        // Do thing B
        break;
    default:
        // Do thing C
}
```

Loops

- `for` loops are used to do something a certain number of times:

```
for (int i = 1; i <= 10; i++) {
    // Do something in here
}
```

- `while` loops are used to do something while a condition is true:

```
while (i < 5) {
    i++;
}
```

```
do {
  i++;
} while (i < 5);
```

- Can use `continue` and `break` for additional control

Arrays and Strings

- An **array** is used to store an indexed group of values of the same type:

```
float temperatures[3];

int studentAges[] = {19, 20, 19, 18, 21};
```

- Items in an array are numbered from 0:

```
temperatures[0] = 20.9;  // Set 1st value
temperatures[1] = 18.7;  // Set 2nd value
temperatures[2] = 19.3;  // Set 3rd value
```

- A **string** is an array of characters, ending with a null character (`\0`)

Characters and Strings

- Don’t confuse C and Bash!

- Single quotes are used for **character literals:**

```
char gradeLetter = 'A';
```

- Some special characters must be written using **escape sequences**:

```
char newline = '\n';
```

- Double quotes are used for **string literals**:

```
char[] animal = "Rabbit";
```

Working with Strings

- Useful string functions contained in `string.h`:

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

- Return length of string: `strlen(string)`
- Compare two strings: `strcmp(str1, str2)`
- Append `str1` to `str2`: `strcat(str2, str1)`
- Copy `str1` to `str2`: `strcpy(str2, str1)`

Pointers

- A **pointer** is a variable which holds the address of a memory location containing another variable

- Create a pointer using an asterisk (`*`):

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

- Assign an address to a pointer using an ampersand (`&`):

```
int x = 12;
int *p = &x;  // p now points to x
```

- Get the value of the variables pointed to by a pointer:

```
printf("%d", *p);
```

Input and Output

- Input and output functions are in **`stdio.h`**:

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

- Display output using **`printf`**:

```
printf("The value of x is %lu.\n", x);
```

- Remember to use the correct format specifiers （**`-Wall`** can help!)

Input and Output

- Get interactive keyboard input from the user:

```
char[50] line;  // Buffer for user input
double freq;

// Prompt user for input
printf("Enter frequency as a double:\n");

// Read a line of input
fgets(line, sizeof(line), stdin);

// Extract double value and store in freq
sscanf(line, "%lf", &freq);
```

Structured Data Types

- A **struct** is used to store a group of named values, possibly of different types:

```
struct point {
    double x, y;
    char *label;
};

typedef struct point point_t;

point_t location = {1.0, 3.0};
```

- Access individual elements:

```
location.y = location.x * 2.0;
```

Functions

- Functions let you give a name to a block of code which can then be used in multiple places

return type (or `void` for none) **name** **parameters (or `void` for none)**

```
double calc_area(double radius) {
    // Calculate area of a circle
    const double pi = 3.14159;
    return pi * radius * radius;
}
```

- Call a function using its name:

```
double a = calc_area(12.0);
```

Command Line Arguments

- Programs can be passed arguments on the command line:

```
tail -n 5 readme.txt
```

arguments

- Use special form of the **`main`** function to get at arguments:

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

- **`argc`** is the number of arguments, including the name of the program itself

- **`argv`** contains the arguments as strings (**`argv[0]`** is the program name, **`argv[1]`** is the first argument, etc.)

File Input and Output: Text

- Input and output functions are in **`stdio.h`**

- Open a file pointer:

```
FILE *f = fopen("MyFile.txt", "r+");
```

- Write to or read from a file:

```
fprintf(f, "%d is a number\n", num);
fscanf(f, "%d is a number", &num);
```

- Flush and close a file after use:

```
fflush(f);
fclose(f);
```

File Input and Output: Binary

- Similar to text, except append **`b`** to the second parameter when opening the file and use **`fwrite`** instead of **`fprintf`**:

```
FILE *f = fopen("MyFile.dat", "wb");

int num = 12;
fwrite(&num, sizeof(int), 1, f);

fflush(f);
fclose(f);
```

Pseudorandom Number Generation

- Functions for PRNGs are in **`stdlib.h`**

- Seed the PRNG with an integer:

```
srand(42);
```

- Get the next random integer:

```
int myRandomNumber = rand();
```

- Can use current time to get a different seed each time you run the program (must also include **`time.h`**):

```
srand(time(NULL));
```

Pseudorandom Number Generation

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

int main(void) {
    // Seed PRNG with current time
    srand(time(NULL));

    // Generate 100 random integers
    for (int n = 0; n < 100; n++) {
        printf("%d\n", rand());
    }

    return 0;
}
```

The Preprocessor

- Include header files with definitions that we need:

```
#include <name of system header>
#include "name of local header"
```

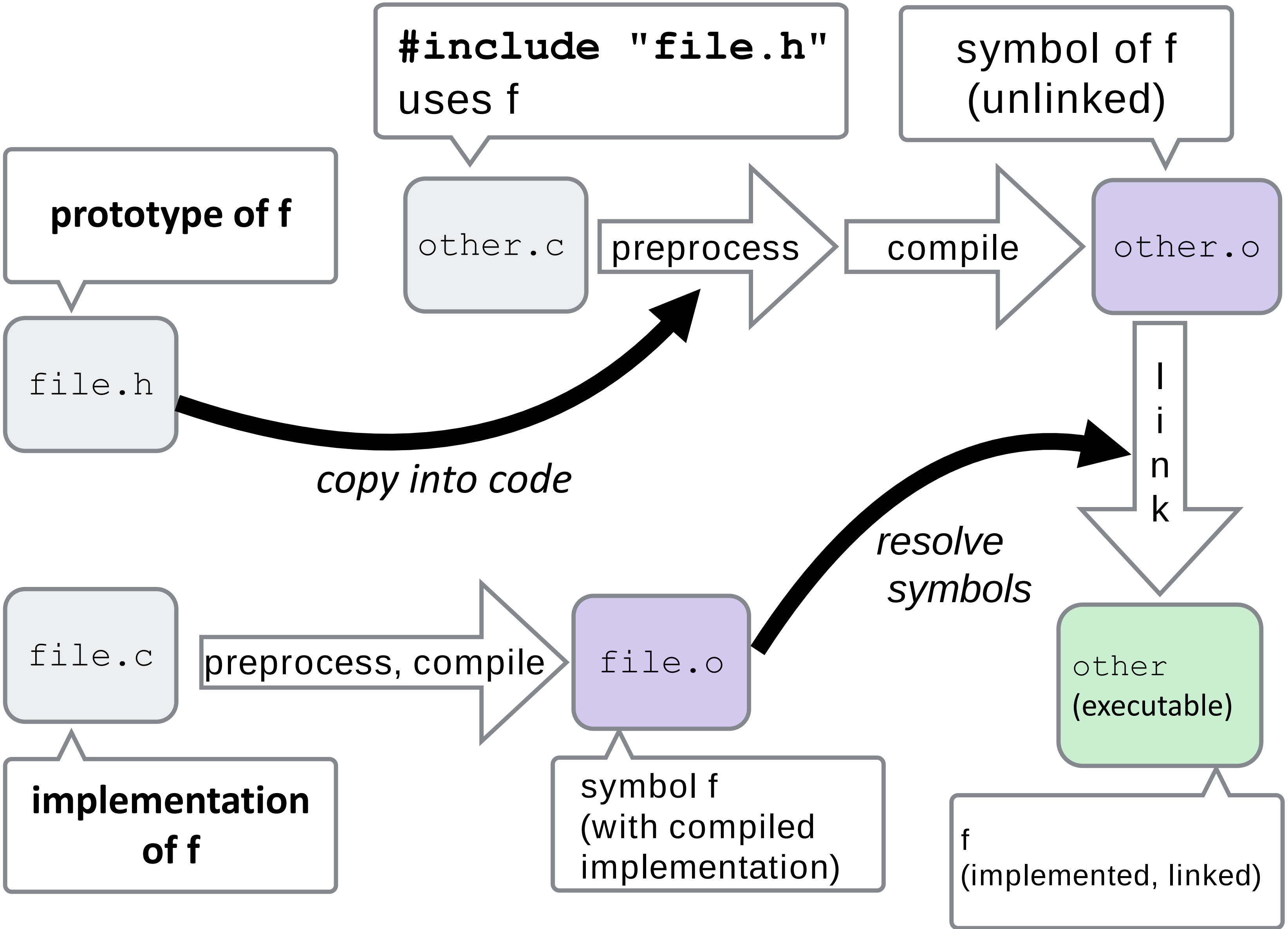
- Can replace **macros** with some text (“find and replace”):

```
#define MYOFFICE 427A
```

- Can conditionally include sections of code:

```
#ifdef SOMETHING
    Code which depends on SOMETHING
#else
    Code if SOMETHING not defined
#endif
```

Compiling and Linking



`#include "file.h"`
uses f

symbol of f
(unlinked)

prototype of f

`other.c`

preprocess

compile

`other.o`

`file.h`

link

copy into code

resolve symbols

`file.c`

preprocess, compile

`file.o`

`other`
(executable)

implementation of f

symbol f
(with compiled implementation)

f
(implemented, linked)