CS 101: Computer Programming and Utilization

12-Functions

Instructor: Sridhar Iyer
IIT Bombay

Activity – Calculate GCD

Write a program to calculate the GCD of 2 numbers.

Use Euclid's Observation:

- If d divides m, n, then

  d divides m-kn, n, for all integers k.

- Euclid's observation can be repeatedly applied to reduce numbers whose GCD we want to find.

  GCD(m, n) = GCD(m-kn, n)

Euclid's observation - example

GCD(3977,943)
= GCD(3034,943)
= GCD(2091,943)
= GCD(205,943)
= GCD(205, 123)
= GCD(82, 123)
= GCD(82, 41)
= 41

We can shorten this process:

205 = 3977 % 943
123 = 943 % 205
82 = 205 % 123
41 = 123 % 82
0 = 82 % 41

Write a program to calculate the GCD of 2 numbers.

GCD program

```
int main() {
  int Large, Small, Remainder;
  cin >> Large >> Small;
  while (true) {
    Remainder = Large % Small;
    if (Remainder == 0) break;
    Large = Small;    //Note this step!
    Small = Remainder;
  } cout << "The GCD is: " << Small << endl;
return 0;
}
```

Repeated calculation of GCD

If we have to evaluate GCD for values (x1,y1) at one place in our program and assign the result to z1,

and evaluate GCD for values (x2, y2) at another place in our program and assign the results to z2.

One way of doing this is to repeat the GCD program code wherever required. → Why is this a bad idea?

It would be useful to write the code once and write statements like z1 = gcd(x1,y1) in the main program.

Functions

A mechanism to write instructions once and reuse them, is provided by most programming languages

- Called a function

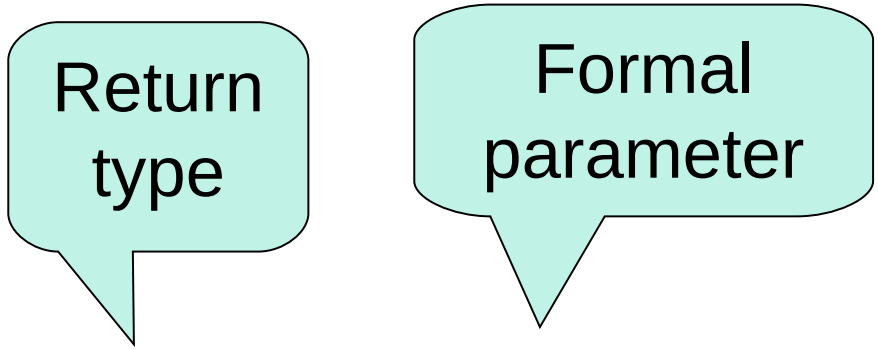
A function accepts ‘parameters’, executes the 'body' (code) on the given parameters and returns the result.

A function can be invoked from within a program as many times as needed.

- Some functions that we have already used – rand(), abs()

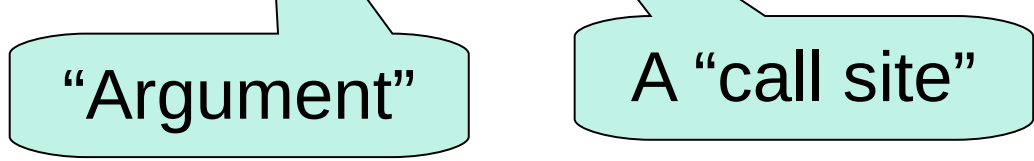
Functions

- Single statement
- Statement block { … }
- If-then-else
- While, for, break, continue
- Next device for writing modular, reusable code: **functions**
  - One declaration
  - Many uses




```
int abs(int a) {
  return a > 0? a : -a;
}
```

```
int x = -3, y = 5;
int z = abs(x) - abs(y);
int b = z*2;
```



Defining functions

```
return-type name-of-function(
                parameter1-type parameter1-name,
                parameter2-type parameter2-name,
                        …) {
            function-body
                        }
```

The type of the value that will be returned by the function. gcd: int

variables for holding values of arguments. gcd: Large, Small

Designer of the function decides the type of the corresponding arguments. gcd: both arguments must be int.

- Definition must appear before use.

GCD function

```
int gcd (int Large, int Small) {
  int Remainder;
  while(true) {
    Remainder = Large % Small;
    if (Remainder == 0) break;
    Large = Small;
    Small = Remainder;
  }
  return Small;
}
```

```
// GCD must be defined or
   declared before main()


int main () {
  int x1, y1, z1, x2, y2, z2;
  cin >> x1 >> y1;
  z1 = gcd (x1, y1);
  cin >> x2 >> y2;
  z2 = gcd (x2, y2);
return 0;
}
```

Function calls

Within a program, a function is invoked simply by using the function name within any expression, with appropriate parameters:

- Main program reaches the function call and suspends.
- Given parameters are copied to the respective locations in the function block.
- Function code is now executed.
- Result value is returned to the main (calling) program, which resumes from where it had suspended.

Difference between functions and threads?

Execution internals

- What happens when you run

```
main() {
  int a = 3;
  fun1(a);
  cout << a;
}?
```

- Just before executing the implementation of fun1, the computer notes down in a special area of RAM what to do after fun1 returns
- fun1 may call fun2 may call fun3…

Recall: Program counter (PC)

- How does the CPU know what to do next?
- Assign IDs (e.g., 1—8 in the example)
- CPU has a register with the ID of the next statement to be executed

- Recall: PC keeps track of statements at the level of assembly instructions, not C++ statements

```
1. cin >> fah;
2. v2 = fah – 32;
3. v3 = v2 / 9;
4. cen = v3 * 5;
5. if (cen < 5) {
6.   cout << “cold”;
7. }
8. kel = cen + 273;
```

A key use of the PC is in implementing function calls

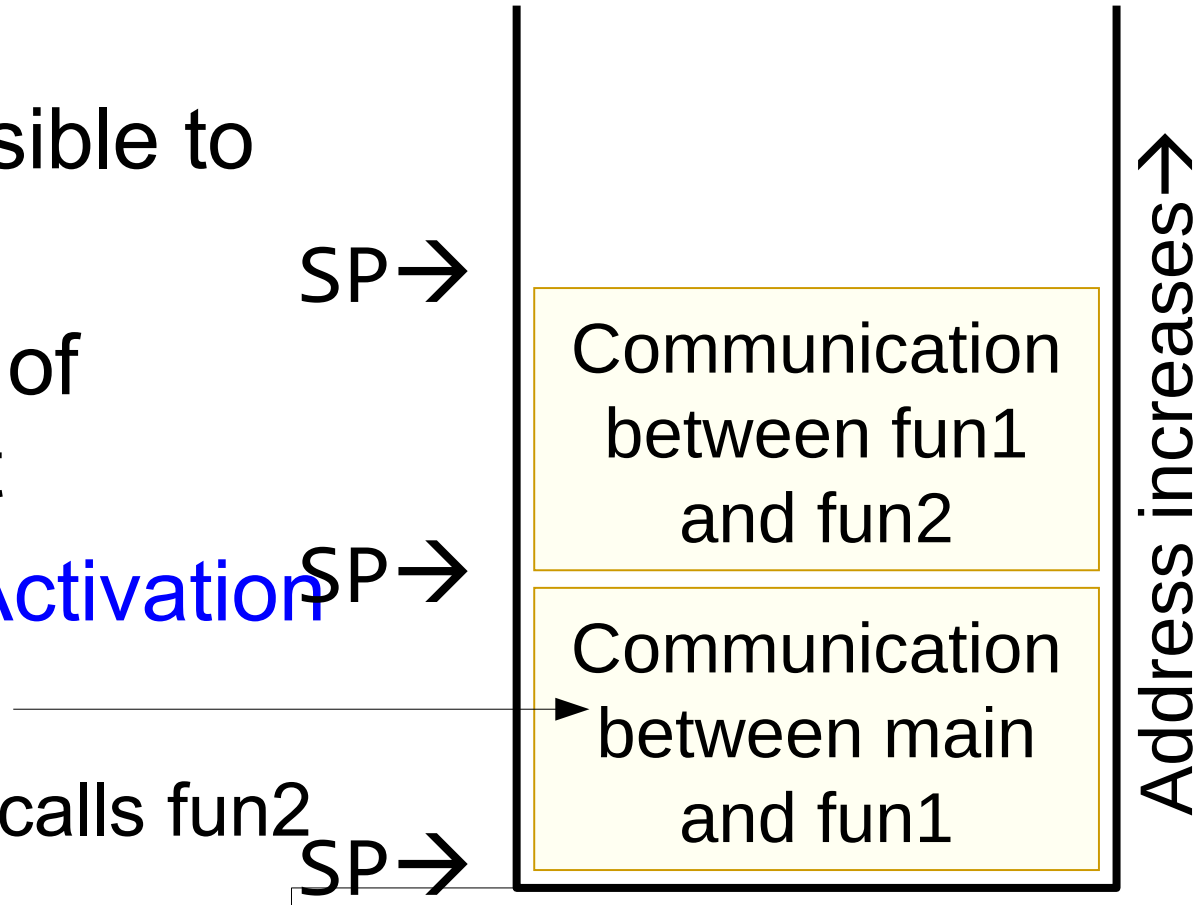
Memory segments

- A process is given three memory segments
- **Code segment**
  - Stores compiled executable code
  - Read-only once loaded
- **Data segment**
  - Storage for variables declared, string, vector<T>, matrix<T>, other data structures
  - Will discuss in more detail later
- **Stack segment**: memory used for
  - Communication between caller and callee
  - Keeping track of pending work

"Segmentation violation"

The stack pointer (SP)

- SP is a special register like PC
- Not directly accessible to the programmer
- Points to the "top" of the stack segment
- Each call has an Activation Record
- e.g., main calls fun1 calls fun2



Base of stack segment, say 4000

See Animated version of this slide, instead of pdf version

Variables and Arguments

New variables can be created in called function.

Set of variables in main() is completely disjoint from the set in called function, gcd.

- Both may contain same name, e.g. Large.

  main() will reference the variables in its scope (activation), and gcd in its activation.

- Change in Large in gcd will not affect Large in main.

Arguments to calls can be expressions, which are first evaluated before called function executes.

Functions can be called while executing functions.

Passing parameters to functions

- Default is “pass by value”

```
void fun(int x) {
x = 5; } // void: does not return anything.
```

```
main() {
   int a = 3;
   fun(a);
   cout << a << endl;
}
```

- Value 3 copied from a to x
- x is locally modified
- Modification has no effect outside fun
- Even if the name 'x' in fun is changed to 'a'

Modifying variables outside function

- Use “pass by reference” instead

```
void fun(int& x) {
X = 5; } // void: does not return anything.
```

```
main() {
   int a = 3;
   fun(a);
   cout << a;
}
```

- No value copy
- x becomes an “alias” for a
- Modifying x amounts to modifying a
- Efficient, especially if large data structures are to be passed across functions

Arrays are passed by reference

```
void readarray (double*m, int n) {
   for(int i=0; i<n; i++) cin >> m[i];
}
int main() { double marks[100];
   readarray(marks,100);
}
```

Array name is copied, so when corresponding parameter is used to access elements, elements of original array are accessed, and they can get modified.

For ordinary variables, value is copied,
so corresponding parameter change does not affect the variable.

Function declarations

- If you declare first, then definitions can come later in any order.
- Useful if you like the main program to come first in the file. Sometimes easier to follow the logic.

```
int gcd(int m, int n); // → Declaration; no body –
int lcm(int m, int n);
int main () { cout << lcm(24,36);}
int lcm(int m, int n){ … } // → Definition; with body
int gcd(int m, int n){ … }
```

says what is the type of gcd:
The full definition will come later possibly in another file.

Note the reuse of m and n!

A function to compute LCM, using GCD

```
int lcm(int m, int n) { return m*n/gcd(m,n); }
// must come after gcd definition
// but before main(), or wherever it is used.

void printlcm(int m, int n) {
   cout << lcm(m,n); return;
} // void: does not return anything.

int main () {int m, n; cin >> m >> n;
printlcm(m,n); return 0;}
```

What is the execution sequence now?

Execution sequence

- main() executes. Suspends when call to printlcm() is encountered. AR of main() is put on the stack. Values of m, n from AR of main are copied into the parameters of printlcm(m, n).
- printlcm() executes. Suspends when call to lcm() encountered. AR of printlcm() put on stack. Values copied for call.
- lcm() executes. Suspends when call to gcd() encountered. AR of lcm() put on stack. Values copied.
- gcd() executes and completes. Result is copied from AR of gcd() to AR of lcm(). AR of gcd() deleted. AR of lcm() taken from stack. Control returns to lcm().
- lcm() resumes and completes. Control returns to printlcm(), which completes and control returns to main().

Separate compilation units

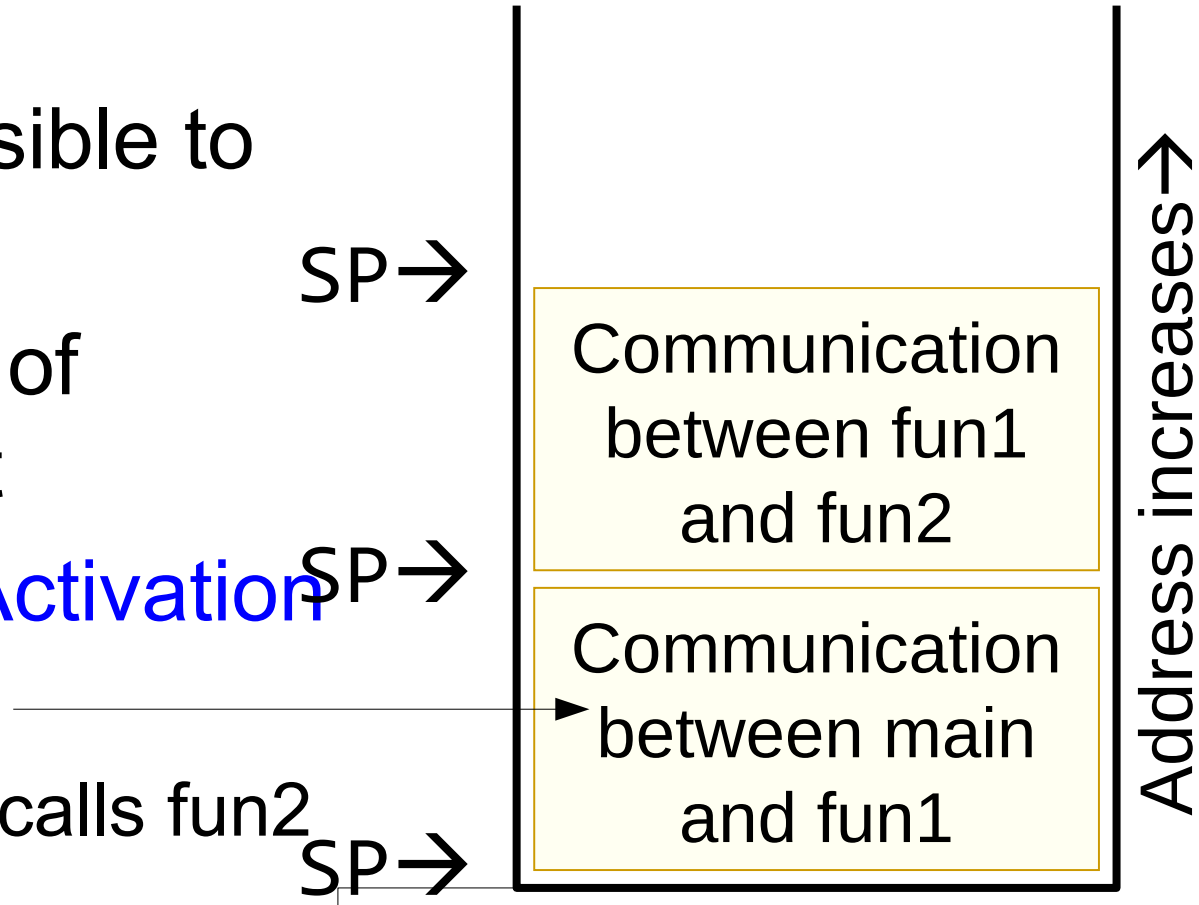
- Thus far, we have placed all our code in one C++ source file.
- Can't share functions from many projects.
- Change one line, compile everything again.
- Solution: The `-c` and `-o` flags to g++

  - Example: Suppose we write the gcd function definition in gcd.cpp and the main program in main.cpp
  - g++ main.cpp gcd.cpp → will produce a.out
  - g++ -c gcd.cpp → will produce gcd.o file
  - g++ -o gcd.o main.o → will produce a.out

Extra Notes
(Optional Reading)

Function call internals

- SP is a special register like PC
- Not directly accessible to the programmer
- Points to the “top” of the stack segment
- Each call has an Activation Record
- e.g., main calls fun1 calls fun2



Base of stack segment, say 4000

See Animated version of this slide instead of pdf versio

Design of the callee

High-level function implementation

```
int abs(int a) {
  return a > 0? a : -a;
}
```

in	Value of a on entry
out	Space to write return value
todo	Code address to jump to on return

Activation Record

Compiled low-level code

abs=2000	a=**stack.top**.in
2001	if a > 0 go to 2003
2002	a = -a
2003	**stack.top**.out = a
2004	go to **stack.top**.todo

Design of the caller

```
main() {
    int x = -3, y = 5;
    int z = abs(x) - abs(y);
    int b = z*2;
}
```

main=1000	x=-3
1001	y=5
1002	push activation record on stack
1003	stack.top.in = x, stack.top.todo=**1005**
1004	go to abs = **2000**
1005	R1 = stack.top.out
1006	pop activation record from stack
1007	push activation record on stack
1008	stack.top.in = y, stack.top.todo=**1010**
1009	go to abs = **2000**
1010	R2 = stack.top.out
1011	pop activation record from stack
1012	z = R1 – R2
1013	b = z * 2

Saving register values

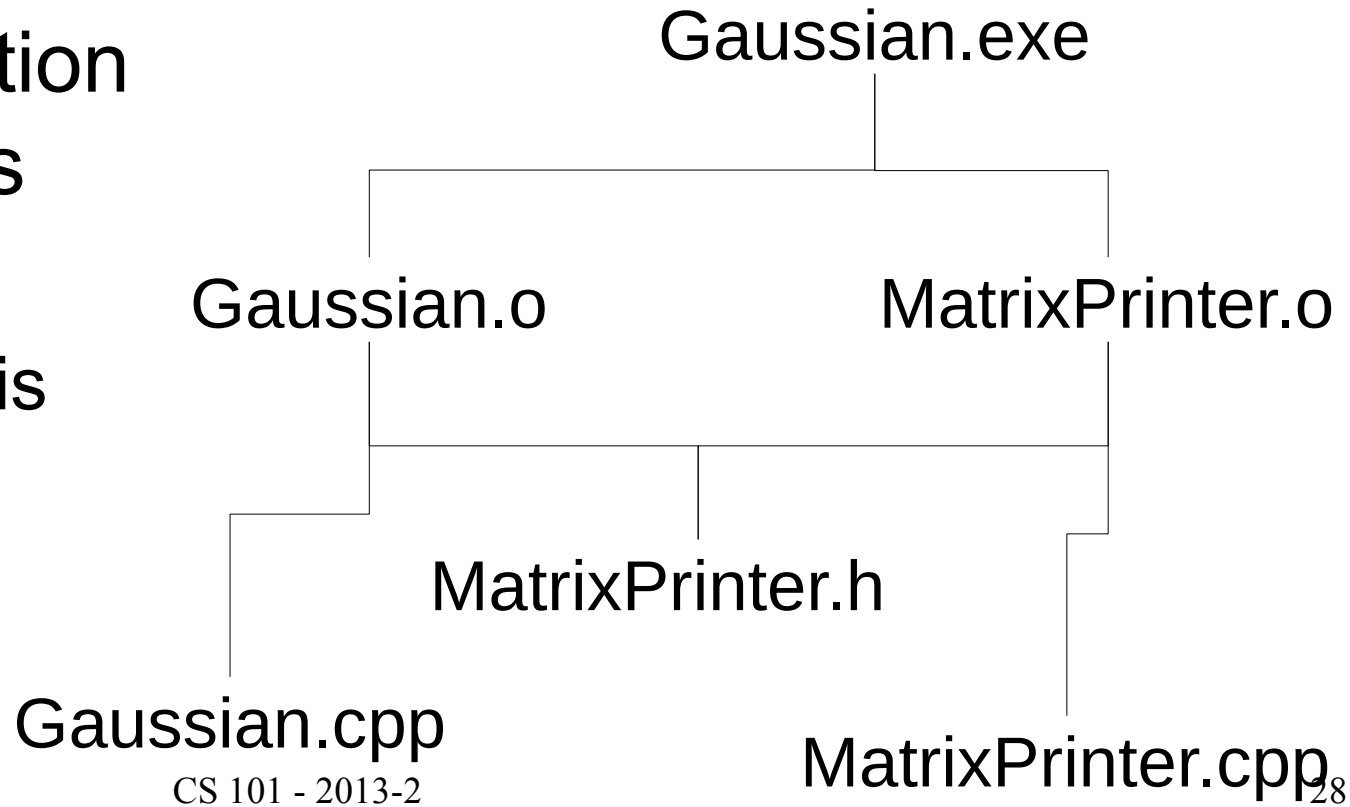
- Suspend caller, execute callee, resume caller
- CPU has a fixed, small set of registers

- Callee must restore registers to values in caller just before callee was invoked
- Also allocate space in activation record to save registers before call
- After call, copy values back to registers before resuming caller

- Overheads of function calls

Compiling multiple files using “make”

- When we change one file in a large project, which files need to be compiled again?
- (File) system provides no consistency check between x.cpp and x.o and a.out
- A *makefile* gives a formal specification of dependencies
- Example:
  - MatrixPrinter.h is used with both cpp files

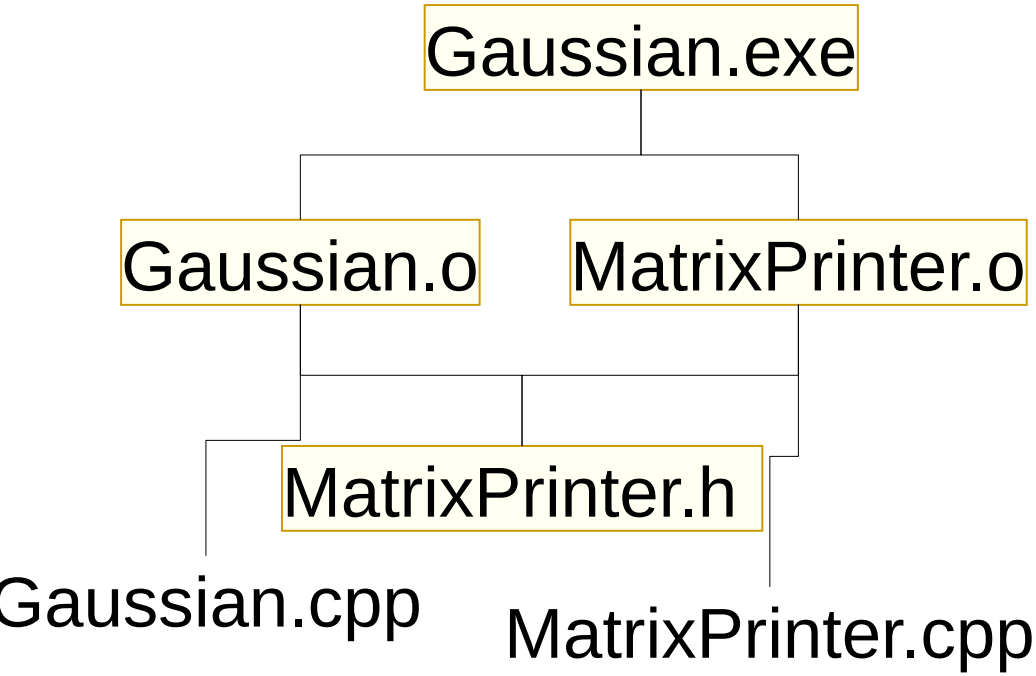


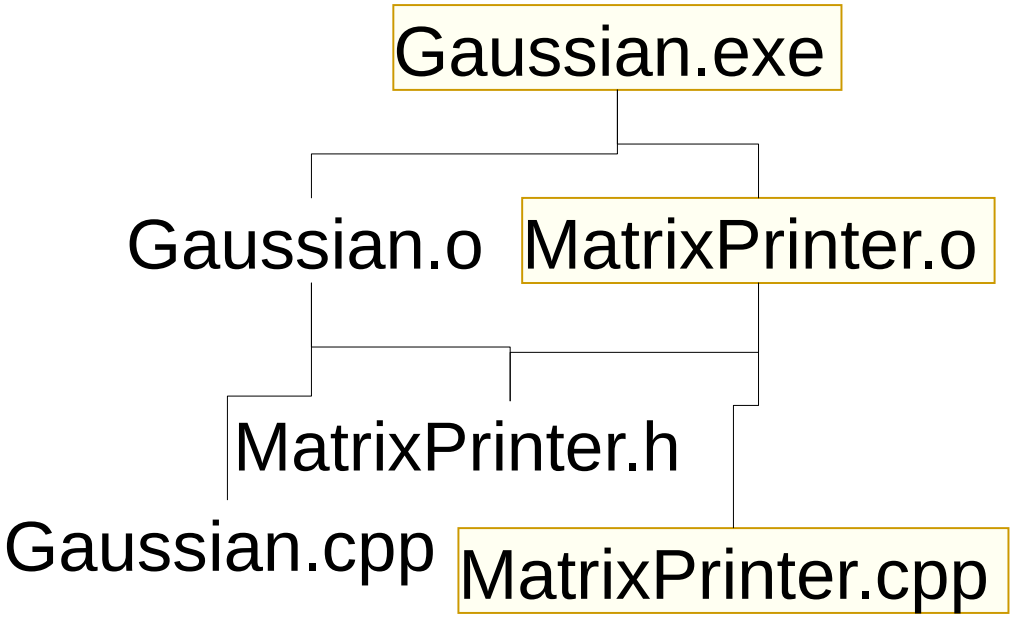
Makefile

- The `make` program looks for a file called `Makefile` or `makefile`
- A makefile consists of **rules** and **actions**
- A rule has the form
  `target: dependencyList`
- An action is a program to run, with command-line arguments

- If any file on the dependency list is newer than the target, (re) run the action
  - Check out the “Build” button in Geany

Make schedules “wavefront” of updates





If MatrixPrinter.h is updated, then the dependent files – Gaussian.o, MatrixPrinter.o and a.out are also re-created.
If MatrixPrinter.cpp is updated, then only MatrixPrinter.o and a.out are re-created.