# Systems Programming Laboratory, Spring 2022

## Introduction to gcc

**Abhijit Das**
**Arobinda Gupta**

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

January 11, 2022

# What you do not know about gcc

- What does the gcc compiler do?
- What are header files? Why should one #include them?
- Why should programs with math functions be compiled with the –*lm* flag?
- What are the compile-time options for gcc?
- How can C programs communicate with the shell?
- What is the C preprocessor?
- How can one write a program in multiple input files?
- What are libraries?
- How can one write one's own libraries?

# The four-stage compilation process

**Preprocessing** This involves the processing of the # directives. Examples:

- The #include'd files are inserted in your code.
- The #define'd macros are literally substituted throughout your code.

**Compiling** The input to this process is the preprocessed C file, and the output is an assembly-language code targeted to the architecture of your machine.

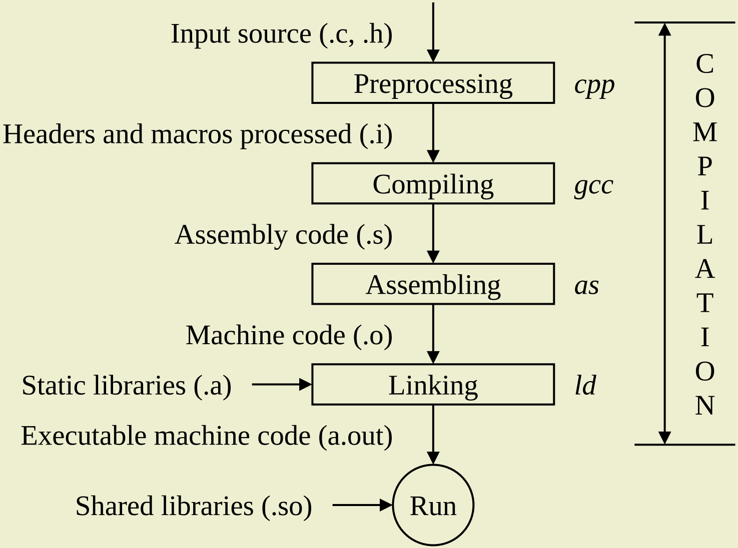
**Assembling** The assembly-language code generated by compiling is converted to a machine code called the object file. The external functions (like printf and sqrt) are still undefined.

**Linking** The object file(s) is/are eventually converted to an executable file in this process. At this point, the external functions from C runtime library and other libraries are included in the executable file.

---

**Loading** Some functions available in shared (or dynamic) libraries are loaded during runtime from shared object files.

# The compilation process in a nutshell



# An example of the four-stage compilation process

## The file demo.c

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

#define TEN 10
#define TWENTY 20

int main ( )
{
    int a, b, c;

    a = TEN;
    b = a + TWENTY;
    c = a * b;
    printf("c = %d\n", c);
    exit(0);
}
```

# Preprocessing

The C preprocessor is called *cpp*.

```
$ cpp demo.c > demo.i
$ cat demo.i
...
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
...
# 7 "demo.c"
int main ( )
{
   int a, b, c;

   a = 10;
   b = a + 20;
   c = a * b;
   printf("c = %d\n", c);
   exit(0);
}
$
```

# Compiling

This needs invoking gcc with the –S flag. A file with extension .s is generated.

```
$ gcc -S demo.i
$ cat demo.s
        .file   "demo.c"
        .text
        .section        .rodata
.LC0:
        .string "c = %d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB6:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $10, -12(%rbp)
        movl    -12(%rbp), %eax
        addl    $20, %eax
        movl    %eax, -8(%rbp)
        movl    -12(%rbp), %eax
        imull   -8(%rbp), %eax
```

```
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %edi
        call    exit@PLT
        .cfi_endproc
...
$
```

# Assembling

- The assembler is called *as*.
- The symbols in object files are listed by *nm*.

```
$ as demo.s -o demo.o
$ nm demo.o
                 U exit
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U printf
```

- printf and exit are undefined in this object file.

# Linking

- This is done by *ld*.
- This requires many libraries and is complicated.
- gcc does it transparently for you.

```
$ gcc demo.o
$ ./a.out
c = 300
$ nm a.out
...
```

- You get a big list of defined symbols.
- printf and exit are still left undefined.

```
...
                 U exit@@GLIBC_2.2.5
...
                 U printf@@GLIBC_2.2.5
...
```

# Runtime loading

- printf and exit are loaded from shared object(s) during runtime.

```
$ ldd a.out
        linux-vdso.so.1 (0x00007ffe80ff2000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f98b5e19000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f98b602d000)
$
```

- If you want these functions to be in your executable, compile with the –static flag.
- This creates a huge a.out.
- You can see printf and exit defined in the executable.

```
$ gcc -static demo.o
$ ldd a.out
        not a dynamic executable
$ nm a.out | grep printf
...
0000000000410bb0 T printf
...
$
```

# Break your code across multiple files

- Modular programming is a good practice, and is needed in any large coding project.
- Large source files take huge time for recompilation.
- If the code is broken down in pieces, then only the pieces that are changed need recompilation.
- Large software development is a two-stage process.
  - Generate object files from individual modules.
  - Merge the object files into a single executable file.
- Sometimes object files are combined in the form of libraries.
- User programs can use the functions archived in libraries during future developments.

# Staque: A multi-file stack-queue application

- We build linked-list implementations of the stack and queue data structures.
- We write the following files.

| | |
|---|---|
| **defs.h** | Defines a node data type. |
| **stack.h** | Defines the stack data type and the stack function prototypes. |
| **queue.h** | Defines the queue data type and the queue function prototypes. |
| **stack.c** | The implementations of the stack functions. |
| **queue.c** | The implementations of the queue functions. |
| **staquecheck.c** | A sample application with the *main* function. |

# The header file defs.h

- Both stacks and queues use nodes defined as follows.

```
typedef struct _node {
   int data;
   struct _node *next;
} node;

typedef node *nodep;
```

- Write these data-type definitions in **defs.h**.

# The header file stack.h

```
typedef nodep stack;                    // Pointer to the beginning of the linked list

stack initstack ( ) ;                   // Create a new empty stack
int emptystack ( stack ) ;              // Check whether the input stack is empty
int top ( stack ) ;                     // Return the top of a stack (if non-empty)
stack push ( stack , int ) ;            // Push an integer to a stack
stack pop ( stack ) ;                   // Pop from a (non-empty) stack
void printstack ( stack ) ;             // Print the elements of a stack from top to bottom
stack destroystack ( stack ) ;          // Delete all the nodes from a stack
```

# The header file queue.h

```
typedef struct {
   nodep front;                          // Pointer to the beginning of the linked list
   nodep back;                           // Pointer to the end of the linked list
} queue;

queue initqueue ( ) ;                    // Create a new empty queue
int emptyqueue ( queue ) ;               // Check whether a queue is empty
int front ( queue ) ;                    // Return the element at the front of a queue (if non-empty)
queue enqueue ( queue , int ) ;          // Insert an integer at the front of a queue
queue dequeue ( queue ) ;                // Delete an element from the back of a (non-empty) queue
void printqueue ( queue ) ;              // Print the elements of a queue from front to back
queue destroyqueue ( queue ) ;           // Delete all the nodes from a queue
```

# The file stack.c

```
#include <stdio.h>
#include <stdlib.h>
#include "defs.h"
#include "stack.h"

stack initstack ( )
{
    stack S;
    S = (stack)malloc(sizeof(node));
    S -> data = 0; S -> next = NULL;
    return S;
}
...
stack destroystack ( stack S )
{
    node *p;
    while (S) {
        p = S; S = S -> next; free(p);
    }
    return NULL;
}
```

# The file queue.c

```
#include <stdio.h>
#include <stdlib.h>
#include "defs.h"
#include "queue.h"

queue initqueue ( )
{
   queue Q;
   node *p;
   p = (node *)malloc(sizeof(node));
   p -> data = 0;
   p -> next = NULL;
   Q.front = Q.back = p;
   return Q;
}
...
queue destroyqueue ( queue Q )
{
   node *p;
   while (Q.front) {
      p = Q.front;
      Q.front = (Q.front) -> next;
      free(p);
   }
   Q.front = Q.back = NULL;
   return Q;
}
```

# The application staquecheck.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "defs.h"
#include "stack.h"
#include "queue.h"

#define ITER_CNT 10

int main ( )
{
    stack S;
    queue Q;
    int i;
    S = initstack();
    for (i=0; i<ITER_CNT; ++i) { S = push(S, rand() % 100); printstack(S); }
    S = destroystack(S);

    Q = initqueue();
    for (i=0; i<ITER_CNT; ++i) { Q = enqueue(Q, rand() % 100); printqueue(Q); }
    Q = destroyqueue(Q);

    exit(0);
}
```

# Compile in one shot

```
$ gcc -Wall staquecheck.c stack.c queue.c
$ ls -l
total 48
-rwxr-xr-x 1 abhij abhij 17640 Dec 23 20:40 a.out
-rw-r--r-- 1 abhij abhij   152 Dec 23 19:43 defs.h
-rw-r--r-- 1 abhij abhij  1262 Dec 23 19:45 queue.c
-rw-r--r-- 1 abhij abhij   360 Dec 23 19:43 queue.h
-rw-r--r-- 1 abhij abhij  1098 Dec 23 19:45 stack.c
-rw-r--r-- 1 abhij abhij   315 Dec 23 19:43 stack.h
-rw-r--r-- 1 abhij abhij   983 Dec 23 20:34 staquecheck.c
$ ./a.out
...
$
```

- The option **`-Wall`** generates most of the relevant warning messages.
- Instead of **`a.out`**, you can generate an executable file of any name by the **`-o`** option.

```
$ gcc -Wall -o myapp staquecheck.c stack.c queue.c
$ ./myapp
```

- Never forget an executable name after –o. Writing the C source file name after –o will replace the file.

# Generating individual object files

- Compile using the **-c** option.
- Does not require a *main* function.
- This does not generate an executable file (even if *main* is there).

```
$ gcc -Wall -c stack.c
$ gcc -Wall -c queue.c
$ gcc -Wall -o myapp staquecheck.c stack.o queue.o
$ ls -l
-rw-r--r-- 1 abhij abhij   152 Dec 23 19:43 defs.h
-rwxr-xr-x 1 abhij abhij 17640 Dec 23 21:01 myapp
-rw-r--r-- 1 abhij abhij  1262 Dec 23 19:45 queue.c
-rw-r--r-- 1 abhij abhij   360 Dec 23 19:43 queue.h
-rw-r--r-- 1 abhij abhij  3424 Dec 23 21:01 queue.o
-rw-r--r-- 1 abhij abhij  1098 Dec 23 19:45 stack.c
-rw-r--r-- 1 abhij abhij   315 Dec 23 19:43 stack.h
-rw-r--r-- 1 abhij abhij  3248 Dec 23 21:01 stack.o
-rw-r--r-- 1 abhij abhij   983 Dec 23 20:34 staquecheck.c
$ ./myapp
...
$
```

# Difference between #include <. . .> and #include ". . ."

- There are default (system-dependent) directories for C header files.
  - /usr/include
  - /usr/local/include
- Header files residing in non-default directories should be included by the **#include "..."** directive.
- You can add to the list of default include directories by the **-I** option.

```
$ gcc -Wall -c -I. stack.c
$ gcc -Wall -c -I. queue.c
$ gcc -Wall -o myapp -I. staquecheck.c stack.o queue.o
```

- These compilations add the current directory to the list of include directories.
- You can now use **#include <defs.h>**, **#include <stack.h>**, and **#include <queue.h>** in the source codes.

# The environment variable C_INCLUDE_PATH

- You can avoid the **-I** flag if you set **C_INCLUDE_PATH**.
- Multiple directories can be added as a colon-separated list DIR1:DIR2:DIR3:...
- . (the current directory) can be one of these directories.
- In bourne shell, this can be done as:

```
$ export C_INCLUDE_PATH=".:/home/foobar/include:/opt/users/foobar/include"
$
```

- C shell users should do this:

```
% setenv C_INCLUDE_PATH ".:/home/foobar/include:/opt/users/foobar/include"
%
```

# Introduction to libraries

- A library is a pre-compiled archive of object files.
- These can be linked to user codes during compilation or during runtime.
- Example: The math library consists of the following.
  1. Data types: float, double, ...
  2. Functions: pow, sqrt, atan, cosh, abs, ...
  3. Constants: M_PI, M_E, M_LOG2E, M_SQRT2, ...
  4. A precompiled archive of implementations of the math functions.
- You only need the first three items during compilation.
- This is achieved by `#include <math.h>`.
- The precompiled math library (Item 4) is needed for linking to your final executable.
- You specify the option `-lm` for this linking.

# Types of libraries

## Static libraries

- Prefix: *lib*
- Extension: *.a*
- The static math library has the name *libm.a*
- Functions from static libraries are inserted in the executable during linking

## Shared (or dynamic) libraries

- Prefix: *lib*
- Extension: *.so* (may be followed by . and a version number)
- The shared math library has the name *libm.so*
- Functions from shared libraries are not inserted in the executable during linking
- The functions are read from the .so objects during runtime

# Building the static staque library

- We have the files *defs.h*, *stack.h*, *queue.h*, *stack.c*, and *queue.c* as before.
- We want to build the static library *libstaque.a*. This will contain all the stack and queue functions as listed earlier.
- The library is not meant to contain any *main* function.
- Application programs like *staquecheck.c* will contain the *main* functions as needed.

---

- Compile individual source files with the –c option to generate the object files.
- Combine the object files into an archive *libstaque.a* using the command *ar*.

# Generate libstaque.a

```
$ gcc -Wall -c stack.c
$ gcc -Wall -c queue.c
$ ar rcs libstaque.a stack.o queue.o
$ ls -l
-rw-r--r-- 1 abhij abhij  152 Dec 23 19:43 defs.h
-rw-r--r-- 1 abhij abhij 7046 Dec 24 18:25 libstaque.a
-rw-r--r-- 1 abhij abhij 1262 Dec 23 19:45 queue.c
-rw-r--r-- 1 abhij abhij  360 Dec 23 19:43 queue.h
-rw-r--r-- 1 abhij abhij 3424 Dec 24 18:23 queue.o
-rw-r--r-- 1 abhij abhij 1098 Dec 23 19:45 stack.c
-rw-r--r-- 1 abhij abhij  315 Dec 23 19:43 stack.h
-rw-r--r-- 1 abhij abhij 3248 Dec 24 18:23 stack.o
-rw-r--r-- 1 abhij abhij  144 Dec 23 19:43 staque.h
$
```

# What is there in libstaque.a

```
$ nm libstaque.a

stack.o:
00000000000001c9 T destroystack
0000000000000036 T emptystack
                 U free
                 U fwrite
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T initstack
                 U malloc
00000000000000f4 T pop
                 U printf
000000000000016a T printstack
00000000000000a8 T push
                 U putchar
                 U stderr
0000000000000055 T top
```

```
queue.o:
0000000000000144 T dequeue
0000000000000242 T destroyqueue
000000000000004a T emptyqueue
00000000000000dd T enqueue
                 U free
0000000000000076 T front
                 U fwrite
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T initqueue
                 U malloc
                 U printf
00000000000001d6 T printqueue
                 U putchar
                 U stderr
$
```

# How to use the library

- To compile the application program *staquecheck.c* as given earlier.
- Include the header files *defs.h*, *stack.h*, and *queue.h*.
- A straightforward compilation fails.

```
$ gcc -Wall staquecheck.c
/usr/bin/ld: /tmp/ccIr2q5J.o: in function `main':
staquecheck.c:(.text+0x12): undefined reference to `initstack'
/usr/bin/ld: staquecheck.c:(.text+0x57): undefined reference to `push'
/usr/bin/ld: staquecheck.c:(.text+0x67): undefined reference to `printstack'
/usr/bin/ld: staquecheck.c:(.text+0x7d): undefined reference to `destroystack'
/usr/bin/ld: staquecheck.c:(.text+0x8b): undefined reference to `initqueue'
/usr/bin/ld: staquecheck.c:(.text+0xdb): undefined reference to `enqueue'
/usr/bin/ld: staquecheck.c:(.text+0xf6): undefined reference to `printqueue'
/usr/bin/ld: staquecheck.c:(.text+0x113): undefined reference to `destroyqueue'
collect2: error: ld returned 1 exit status
$
```

# How to link the library

- Like *–lm*, you should compile with *–lstaque*.

```
$ gcc -Wall staquecheck.c -lstaque
/usr/bin/ld: cannot find -lstaque
collect2: error: ld returned 1 exit status
$
```

- The linker *ld* does not look in the current directory for searching libraries.
- The *–L* option advises the linker to add directories to the library path.

```
$ gcc -Wall -L. staquecheck.c -lstaque
$ ls -l
-rwxr-xr-x 1 abhij abhij 17536 Dec 24 18:52 a.out
-rw-r--r-- 1 abhij abhij   152 Dec 23 19:43 defs.h
-rw-r--r-- 1 abhij abhij  7046 Dec 24 18:25 libstaque.a
-rw-r--r-- 1 abhij abhij  1262 Dec 23 19:45 queue.c
-rw-r--r-- 1 abhij abhij   360 Dec 23 19:43 queue.h
-rw-r--r-- 1 abhij abhij  3424 Dec 24 18:23 queue.o
-rw-r--r-- 1 abhij abhij  1098 Dec 23 19:45 stack.c
-rw-r--r-- 1 abhij abhij   315 Dec 23 19:43 stack.h
-rw-r--r-- 1 abhij abhij  3248 Dec 24 18:23 stack.o
-rw-r--r-- 1 abhij abhij   473 Dec 24 18:52 staquecheck.c
-rw-r--r-- 1 abhij abhij   144 Dec 23 19:43 staque.h
$
```

# How to avoid –L?

- You do not need *–L* for *–lm*, but why now?
- This is because the math library resides in a standard library directory.
  - /usr/lib
  - /usr/local/lib
  - /usr/lib/x86_64-linux-gnu/
- If you copy *libstaque.a* to a standard directory, you do not need *–L*.
- Also, your application programs can be anywhere in the file system.
- This needs superuser privilege.

```
$ rm a.out
$ sudo cp libstaque.a /usr/local/lib/
[sudo] password for abhij:
$ gcc -Wall staquecheck.c -lstaque
$ ls -l
-rwxr-xr-x 1 abhij abhij 17536 Dec 24 19:07 a.out
...
$
```

# What about the header files?

- The header files may reside in a default directory.
  - Any application can #include $< \cdots >$ them without –*I*.
  - The application programs do not need to know where the header files are.
- There are standard header directories.
  - /usr/include
  - /usr/local/include
- A user with superuser privileges can copy the header files to one of these directories.
- Using subdirectories is a good option.

# Installing the libstaque headers

- Write an outer wrapper *staque.h*.

```
#include <staque/defs.h>
#include <staque/stack.h>
#include <staque/queue.h>
```

- Run the following commands.

```
$ sudo mkdir /usr/include/staque
$ sudo cp defs.h stack.h queue.h /usr/include/staque
$ sudo cp staque.h /usr/include
$
```

- Subsequently, any application program may only `#include <staque.h>`.
- The three required header files are in turn included by this.

# The keyword extern

- The functions declared in the header files are not implemented by your code.
- These functions are implemented in external library/libraries.
- The key word extern directs the compiler to wait for these implementations.

**The header file queue.h**

```
extern queue initqueue ( ) ;
extern int emptyqueue ( queue ) ;
extern int front ( queue ) ;
extern queue enqueue ( queue , int ) ;
extern queue dequeue ( queue ) ;
extern void printqueue ( queue ) ;
extern queue destroyqueue ( queue ) ;
```

# Building the shared staque library

- We again need only the files *defs.h*, *stack.h*, *queue.h*, *stack.c*, and *queue.c*.
- We plan to generate *libstaque.so*.

---

- Compile individual source files with the –c option to generate the object files.
- Use the option *–fPIC* to generate position-independent codes.
- Combine the objects into the shared library using *gcc –shared*.

```
$ gcc -Wall -fPIC -c stack.c
$ gcc -Wall -fPIC -c queue.c
$ gcc -shared -o libstaque.so stack.o queue.o
$ ls -l
...
-rwxr-xr-x 1 abhij abhij 16928 Dec 24 20:51 libstaque.so
...
$
```

- You can `nm libstaque.so` to find all the defined and undefined symbols.

# How to *link* libstaque.so?

- The linker is not supposed to link the stack and queue functions to applications.
- These functions will be read from *libstaque.so* during runtime.
- Again you need the –L option to add the path of the library.
- If you (in the superuser mode) copy *libstaque.so* to a system directory, then you do not need –L.

```
$ sudo cp libstaque.so /usr/local/lib/
$ gcc -Wall staquecheck.c -lstaque
$ ls -l
-rwxr-xr-x 1 abhij abhij 17064 Dec 24 21:05 a.out
...
$
```

# Libstaque functions are undefined in your a.out

```
$ nm a.out | grep " U "
                 U destroyqueue
                 U destroystack
                 U enqueue
                 U exit@@GLIBC_2.2.5
                 U initqueue
                 U initstack
                 U __libc_start_main@@GLIBC_2.2.5
                 U printqueue
                 U printstack
                 U push
                 U putchar@@GLIBC_2.2.5
                 U rand@@GLIBC_2.2.5
$
```

Good, but you still (perhaps) cannot run your a.out.

```
$ ./a.out
./a.out: error while loading shared libraries: libstaque.so: cannot open shared object file: No such file or
directory
$ ldd a.out
        linux-vdso.so.1 (0x00007ffd0b250000)
        libstaque.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb539500000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fb539714000)
$
```

# Set the runtime library path

- You need to set the environment variable **LD_LIBRARY_PATH**.
- If you use the bourne shell, do this:

```
$ export LD_LIBRARY_PATH=/usr/local/lib
```

- If you use the C shell, do this:

```
% setenv LD_LIBRARY_PATH /usr/local/lib
```

- Now, check whether libstaque.so is found.

```
$ ldd a.out
        linux-vdso.so.1 (0x00007ffe643b1000)
        libstaque.so => /usr/local/lib/libstaque.so (0x00007f780a59e000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f780a391000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f780a5aa000)
$
```

- Go ahead, and run your a.out. No complaints from anybody.

# Some useful gcc options

–W –Wall includes the following (among others). Some of these have many subcategories.

–Wcomment Warn about nested comments.

–Wformat Warn about type mismatches in scanf and printf.

–Wunused Warn about unused variables.

–Wimplicit Warn about functions used before declaration.

–Wreturn-type Warn about returning void for functions with non-void return values.

–Wall does not include the following (among otheres).

–Wconversion Warn about implicit type conversions.

–Wshadow Warn about shadowed variables.

–Werror Convert warnings to errors.

# Some useful gcc options (contd.)

- –g Add debugging information in the executable and object files.
- –pg Compile for profiling.
- –O Set the optimization level.
  - –O0 No optimization (default behavior, useful when debugging).
  - –O1, –O2, –O3 Various levels of optimization. Optimization is time-consuming, and can be used only during the last stages of development.
  - –Os Optimize (reduce) the size of the code.
- –v Verbose mode of compilation.
- – –help Print help message for usage.
- – –version Print the gcc version.

# The C preprocessor

- The C preprocessor has the name *cpp*.
- This has two basic jobs.
  - Insert the #include'd files into your code.
  - Processing the macros.
- Macros can be defined in two ways.
  - Using #define in your code.
  - By the command-line option –D.
- Checking whether a macro is defined or not is possible.
- Macros can be parameterized.

# Macros used as flags

`#define MACRONAME` Define the macro MACRONAME.

`#undef MACRONAME` Undefine the macro MACRONAME.

`#ifdef MACRONAME` If MACRONAME is defined.

`#ifndef MACRONAME` If MACRONAME is not defined.

`#else` The beginning of the else block of an #ifdef or an #ifndef.

`#endif` The end of the conditional code.

# Example of macros used as flags

## The file macros.c

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

#define MYFLAG

int main ()
{
   #ifdef MYFLAG
   printf("MYFLAG is defined\n");
   #undef MYFLAG
   #else
   printf("MYFLAG is not defined\n");
   #endif

   #ifndef MYFLAG
   printf("MYFLAG is undefined here\n");
   #else
   printf("MYFLAG is still defined here\n");
   #endif

   exit(0);
}
```

```
$ gcc -Wall macros.c
$ ./a.out
MYFLAG is defined
MYFLAG is undefined here
$
```

Remove `#define MYFLAG` from macros.c.

```
$ gcc -Wall macros.c
$ ./a.out
MYFLAG is not defined
MYFLAG is undefined here
$
```

# Redefine MYFLAG from command line

```
$ gcc -Wall -DMYFLAG macros.c
$ ./a.out
MYFLAG is defined
MYFLAG is undefined here
$
$ cpp -DMYFLAG macros.c
...
int main ()
{

   printf("MYFLAG is defined\n");


   printf("MYFLAG is undefined here\n");


   exit(0);
}
$
```

- Defining macros using –D offers compilation-time flexibility.
  - You compile your assignments with –DDIAGNOSTIC.
  - Your TA does not require the diagnostic messages, and compiles without this flag.
- On the flip side, some programs may refuse to compile without macros defined.

# Use of macros as flags

- Conditional execution with diagnostic messages (helpful during development).
- Protecting parts of code. The following header file can be #include's multiple times. The flag prevents the stack data type and the function prototypes from getting declared multiple times.

**The header file stack.h**

```
#ifndef __LIBSTAQUE_STACK_H
#define __LIBSTAQUE_STACK_H

typedef nodep stack;

extern stack initstack ( ) ;
extern int emptystack ( stack ) ;
extern int top ( stack ) ;
extern stack push ( stack , int ) ;
extern stack pop ( stack ) ;
extern void printstack ( stack ) ;
extern stack destroystack ( stack ) ;

#endif
```

# Use of macros as values for substitution

## The file macroval.c

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

#define EXPR1 100
#define EXPR2 10 * 10

int main ()
{
   if (EXPR1 == EXPR2) printf("EXPR1 is equal to EXPR2\n");
   else printf("EXPR1 is not equal to EXPR2\n");
   if (EXPR1 == EXPR3) printf("EXPR1 is equal to EXPR3\n");
   else printf("EXPR1 is not equal to EXPR3\n");
   if (EXPR1 == EXPR4 * EXPR4) printf("EXPR1 is equal to EXPR4 * EXPR4\n");
   else printf("EXPR1 is not equal to EXPR4 * EXPR4\n");
   exit(0);
}
```

- This program cannot compile as such, because EXPR3 and EXPR4 are not defined.
- We define these macros by the –D option.

# Examples of macros as values for substitution

```
$ gcc -Wall -DEXPR3="50 + 50" -DEXPR4="5 + 5" macroval.c
$ ./a.out
EXPR1 is equal to EXPR2
EXPR1 is equal to EXPR3
EXPR1 is not equal to EXPR4 * EXPR4
$
```

- Macros are **literally substituted** by the C preprocessor.
- Macros are **not evaluated before** the substitution.
- EXPR4 * EXPR4 is substituted as 5 + 5 * 5 + 5 which evaluates to 35 (not 100).

```
$ cpp -DEXPR3="50 + 50" -DEXPR4="5 + 5" macroval.c
...
int main ()
{
   if (100 == 10 * 10) printf("EXPR1 is equal to EXPR2\n");
   else printf("EXPR1 is not equal to EXPR2\n");
   if (100 == 50 + 50) printf("EXPR1 is equal to EXPR3\n");
   else printf("EXPR1 is not equal to EXPR3\n");
   if (100 == 5 + 5 * 5 + 5) printf("EXPR1 is equal to EXPR4 * EXPR4\n");
   else printf("EXPR1 is not equal to EXPR4 * EXPR4\n");
   exit(0);
}
$
```

# Quoting strings with –D

## The file macrostr.c

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

int main ()
{
   printf("Welcome %s\n", MYNAME);
   exit(0);
}
```

```
$ gcc -DMYNAME="Sad Tijihba" macrostr.c
Many error messages
$ cpp -DMYNAME="Sad Tijihba" macrostr.c
...
int main ()
{
   printf("Welcome %s\n", Sad Tijihba);
   exit(0);
}
$
```

```
$ gcc -DMYNAME='"Sad Tijihba"' macrostr.c
$ ./a.out
Welcome Sad Tijihba
$ cpp -DMYNAME='"Sad Tijihba"' macrostr.c
...
int main ()
{
   printf("Welcome %s\n", "Sad Tijihba");
   exit(0);
}
$
```

# Appendix

# Talking with the shell

- You run your compiled executable (like a.out) from the shell.
- You may add one or more command-line arguments.
- These arguments should somehow go to your C program.
- When the program finishes execution, it should return something to the shell.
- The return value conventionally indicates successful/unsuccessful termination.

**The fully decorated main() function**

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

# The shell talks to your program

- argc is the count of arguments including the program name (like ./a.out).
- argv is a null-terminated array of strings storing the command-line arguments.
- Each argument is a string.
- Use the library functions *atoi*, *atol*, *atof*, . . . (defined in stdlib.h) to convert arguments to int, long int, double, . . . .
- For example, if you run `./a.out 2022 -name "Sad Tijihba" 6.32`, then we have
  - `argc` = 5,
  - `argv[0]` = `"./a.out"`,
  - `argv[1]` = `"2022"`,
  - `argv[2]` = `"-name"`,
  - `argv[3]` = `"Sad Tijihba"`,
  - `argv[4]` = `"6.32"`, and
  - `argv[5]` = NULL.

# Your program talks to the shell

- The return type of main is int.
- You use return or exit() to pass a value to the shell.
- Only an integer value can be returned.
- Conventionally, the return value is an indicator whether the program completed successfully or not.
- 0 means successful termination.
- Any non-zero return value means unsuccessful termination.

# An example chat

## The file circle.c

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

int main ( int argc, char *argv[] )
{
   int c, d, r, x, y, t1, t2, t3;
   char s1, s2, s3;

   if (argc != 6) {
      fprintf(stderr, "*** Incorrect number of arguments\n");
      exit(1);
   }

   c = atoi(argv[1]); d = atoi(argv[2]); r = atoi(argv[3]);
   x = atoi(argv[4]); y = atoi(argv[5]);

   t1 = -2*c; s1 = (t1 >= 0) ? '+' : '-'; if (t1 < 0) t1 = -t1;
   t2 = -2*d; s2 = (t2 >= 0) ? '+' : '-'; if (t2 < 0) t2 = -t2;
   t3 = c*c + d*d - r*r; s3 = (t3 >= 0) ? '+' : '-'; if (t3 < 0) t3 = -t3;

   printf("The equation of the circle:  x^2 + y^2 %c %dx %c %dy %c %d = 0\n", s1, t1, s2, t2, s3, t3);
   if ((x - c) * (x - c) + (y - d) * (y - d) <= r * r) printf("(%d,%d) is inside the circle\n", x, y);
   else printf("(%d,%d) is outside the circle\n", x, y);

   exit(0);
}
```