ECE 264 Spring 2023

Advanced C Programming

Aravind Machiry
Purdue University

This class has more than 400 students and 18 assignments. Everything is automated.

Everyone wants you to get A.
Please help everyone.

Grading Programming Assignments

- Some test cases will be provided to you.
- Some additional test cases may be used during grading.
- “Correct outputs” are only part of the scores.
- ***Your submissions are graded by computer programs. Nothing will be entered by keyboard.***
- Your programs **must not** have gcc warnings or leak memory.
- Your programs **must not** have unwanted messages.

This class will give as many partial credits as possible. However, it is sometimes impossible.

When are partial credits not possible?

- If you do not submit anything
- If you do not submit all needed files
- If your submission cannot compile
- If you modify one file that must not be modified
- If you have erroneous code outside #ifdef and #endif

Please spend a few minutes checking whether you submit all needed files

Your scores depend on **ONLY** your submissions. Nothing else.

Your scores depend on your submissions

- Your scores do ***not*** depend on
- what is stored in your computer
- how much time you spend
- how much you love the class
- It is ***strictly forbidden*** to see the files in students' computers for grading.
- It is ***strictly forbidden*** to modify anything in your submissions for grading.

In the past, some students requested higher scores based on these reasons.

How can you save your precious time?

Case 1:

- Spend 7 hours doing homework
- **Spend 30 seconds submitting**
- **Forget one needed file**
- Receive 0 in this assignment
- Spend 3 hours sending emails to instructor, department head, dean, provost, Purdue president requesting regrading

⇒ 10 hours, 0 point

Case 2:

- Spend 7 hours doing homework
- **Spend 3 minutes submitting (tag ‘final_ver’)**
- Submit all needed files
- Receive a high score

⇒ 7 hours + 3 minutes, high score

Everyone wants you to get A.
Please help everyone.

argc and argv

Command line arguments

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *  * argv)
{
    int ind;
    printf("argc = %d\n", argc);
    for (ind = 0; ind < argc; ind ++)
    {
        printf("argv[%d] = %s\n", ind, argv[ind]);
    }
    return EXIT_SUCCESS;
}
```

Command line arguments

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
  int ind;
  printf("argc = %d\n", argc);
  for (ind = 0; ind < argc; ind ++)
  {
    printf("argv[%d] = %s\n", ind, argv[ind]);
  }
  return EXIT_SUCCESS;
}
```

ind is 0, 1, 2, ... argc - 1

Command line arguments

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
    int ind;
    printf("argc = %d\n", argc);
    for (ind = 0; ind < argc; ind ++)
    {
        printf("argv[%d] = %s\n", ind, argv[ind]);
    }
    return EXIT_SUCCESS;
}
```

ind is 0, 1, 2, ... argc - 1

print the index

and the value of the argument

Using command line arguments 1

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
        if (argc < 2)
        {
                printf("Need a number\n");
                return EXIT_FAILURE;
        }
        int val = strtol(argv[1], NULL, 10);
        val += 10;
        printf("argv[1] = %s\n", argv[1]);
        printf("val = %d\n", val);
        return EXIT_SUCCESS;
}
```

Using command line arguments 1

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * * argv)
{
        if (argc < 2)
        {
                printf("Need a number\n");
                return EXIT_FAILURE;
        }
        int val = strtol(argv[1], NULL, 10);
        val += 10;
        printf("argv[1] = %s\n", argv[1]);
        printf("val = %d\n", val);
        return EXIT_SUCCESS;
}
```

Make sure to check the value

Using command line arguments 2

```
#include <string.h>
int main(int argc, char * * argv)
{
  if (argc < 4)
    {
      printf("Need three arguments\n");
      return EXIT_FAILURE;
    }
  int val1 = strtol(argv[1], NULL, 10);
  int val2 = strtol(argv[2], NULL, 10);
  if (strcmp(argv[3], "+") == 0)
    {
      printf("%d + %d = %d\n", val1, val2, val1 + val2);
    }
```

Using command line arguments 2

```
#include <string.h>
int main(int argc, char * * argv)
{
  if (argc < 4)
    {
      printf("Need three arguments\n");
      return EXIT_FAILURE;
    }
  int val1 = strtol(argv[1], NULL, 10);
  int val2 = strtol(argv[2], NULL, 10);
  if (strcmp(argv[3], "+") == 0)
    {
      printf("%d + %d = %d\n", val1, val2, val1 + val2);
    }
```

convert string to integer

Using command line arguments 2

```
#include <string.h>
int main(int argc, char * * argv)
{
  if (argc < 4)
    {
      printf("Need three arguments\n");
      return EXIT_FAILURE;
    }
  int val1 = strtol(argv[1], NULL, 10);
  int val2 = strtol(argv[2], NULL, 10);
  if (strcmp(argv[3], "+") == 0)
    {
      printf("%d + %d = %d\n", val1, val2, val1 + val2);
    }
```

compare two strings

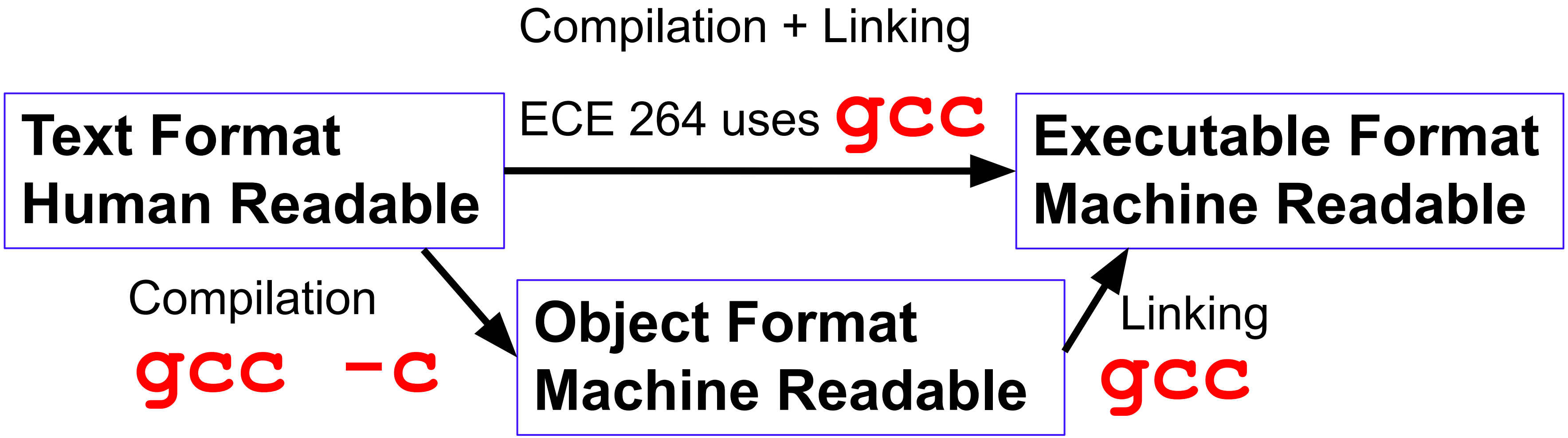
Using command line arguments 2

```
#include <string.h>
int main(int argc, char * * argv)
{
  if (argc < 4)
    {
      printf("Need three arguments\n");
      return EXIT_FAILURE;
    }
  int val1 = strtol(argv[1], NULL, 10);
  int val2 = strtol(argv[2], NULL, 10);
  if (strcmp(argv[3], "+") == 0)
    {
      printf("%d + %d = %d\n", val1, val2, val1 + val2);
    }
```

print the sum

Makefiles

C Programs has three formats



Compilation + Linking

ECE 264 uses `gcc`

Text Format
Human Readable

Executable Format
Machine Readable

Compilation

`gcc -c`

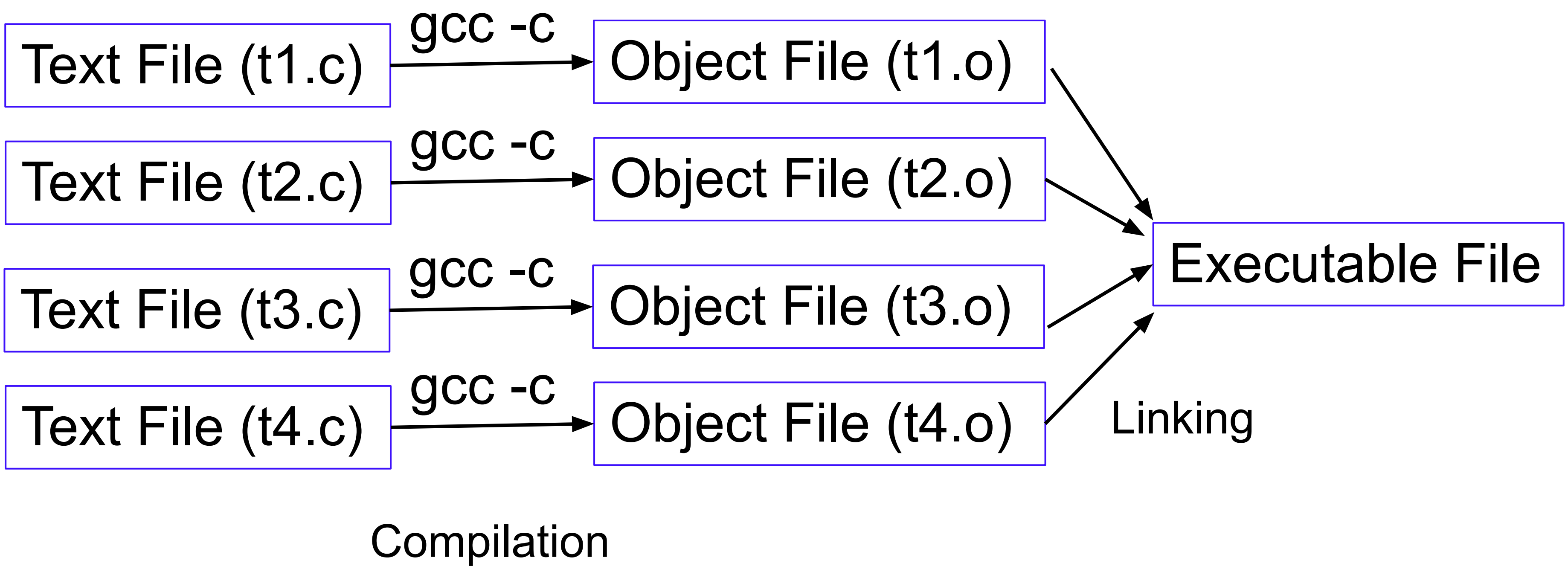
Object Format
Machine Readable

Linking

`gcc`

These formats allow the same programs (text format) to run on different types of machines.

C Programs has three formats



Text File (t1.c) → gcc -c → Object File (t1.o)

Text File (t2.c) → gcc -c → Object File (t2.o)

Text File (t3.c) → gcc -c → Object File (t3.o)

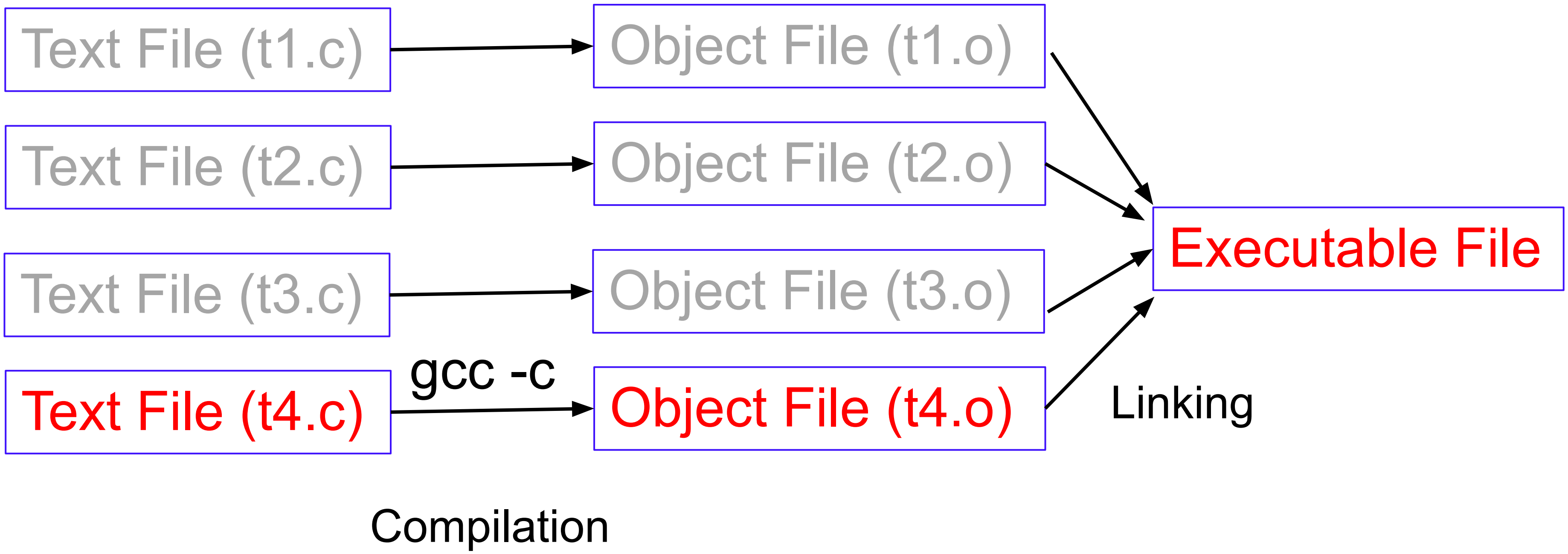
Text File (t4.c) → gcc -c → Object File (t4.o)

Executable File

Linking

Compilation

C Programs has three formats



Text File (t1.c)

Text File (t2.c)

Text File (t3.c)

Text File (t4.c)

gcc -c

Object File (t1.o)

Object File (t2.o)

Object File (t3.o)

Object File (t4.o)

Executable File

Linking

Compilation

Two-Stage process to create executable

- gcc should always have the warnings turned on
- keep track of which .c files have been changed
- compile all changed .c files to generate .o files
- link .o files to create executable

Two-Stage process to create executable

- gcc should always have the warning turned on
- keep track of which .c files have been changed
- compile all changed .c files to generate .o files
- link .o files to create executable

This is a lot of work.
Fortunately, you can use Makefile.

Makefile Introduction

- Need for targets.
- Dependencies.

Makefile Introduction

```
# This is a simple Makefile

target1:
	echo "Hello World\n"

target2: target1
	echo "ECE264"
```

Makefile Introduction: Targets

```
# This is a simple Makefile

target1:
	echo "Hello World\n"

target2: target1
	echo "ECE264"
```

Makefile Introduction: Dependency

```
# This is a simple Makefile

target1:
	echo "Hello World\n"

target2: target1
	echo "ECE264"
```

Simple Makefile

```
# Simple makefile: Specifying all targets manually

addprog: main.o add.o
	gcc main.o add.o -o addprog

main.o:
	gcc -c main.c -o main.o

add.o:
	gcc -c add.c -o add.o
```

Final Makefile

```
# Makefile version 3: with all dependencies
WARNINGS = -Wall -Wshadow --pedantic
ERRORS = -Wvla -Werror
GCC = gcc -std=c99 -g $(WARNINGS) $(ERRORS)
SRCS = main.c add.c
OBJS = $(SRCS:%.c=%.o)
addprog: $(OBJS)
	$(GCC) $(OBJS) -o addprog
test1: addprog
	cat inputs/input1 | $<
%.o: %.c
	$(GCC) -c $< -o $@

clean:
	rm $(OBJS) addprog
```

Final Makefile: Using variables

```
# Makefile version 3: with all dependencies
WARNINGS = -Wall -Wshadow --pedantic
ERRORS = -Wvla -Werror
GCC = gcc -std=c99 -g $(WARNINGS) $(ERRORS)
SRCS = main.c add.c
OBJS = $(SRCS:%.c=%.o)
addprog: $(OBJS)
	$(GCC) $(OBJS) -o addprog
test1: addprog
	cat inputs/input1 | $<
%.o: %.c
	$(GCC) -c $< -o $@

clean:
	rm $(OBJS) addprog
```

Final Makefile: Regular expression

```
# Makefile version 3: with all dependencies
WARNINGS = -Wall -Wshadow --pedantic
ERRORS = -Wvla -Werror
GCC = gcc -std=c99 -g $(WARNINGS) $(ERRORS)
SRCS = main.c add.c
OBJS = $(SRCS:%.c=%.o)
addprog: $(OBJS)
	$(GCC) $(OBJS) -o addprog
test1: addprog
	cat inputs/input1 | $<
%.o: %.c
	$(GCC) -c $< -o $@

clean:
	rm $(OBJS) addprog
```

Final Makefile: Matching rules based on regular expression

```
# Makefile version 3: with all dependencies
WARNINGS = -Wall -Wshadow --pedantic
ERRORS = -Wvla -Werror
GCC = gcc -std=c99 -g $(WARNINGS) $(ERRORS)
SRCS = main.c add.c
OBJS = $(SRCS:%.c=%.o)
addprog: $(OBJS)
	$(GCC) $(OBJS) -o addprog
test1: addprog
	cat inputs/input1 | $<
%.o: %.c
	$(GCC) -c $< -o $@

clean:
	rm $(OBJS) addprog
```

Final Makefile: Using special variables

```
# Makefile version 3: with all dependencies
WARNINGS = -Wall -Wshadow --pedantic
ERRORS = -Wvla -Werror
GCC = gcc -std=c99 -g $(WARNINGS) $(ERRORS)
SRCS = main.c add.c
OBJS = $(SRCS:%.c=%.o)
addprog: $(OBJS)
	$(GCC) $(OBJS) -o addprog
test1: addprog
	cat inputs/input1 | $<
%.o: %.c
	$(GCC) -c $< -o $@

clean:
	rm $(OBJS) addprog
```

Final Makefile: Testing

```
# Makefile version 3: with all dependencies
WARNINGS = -Wall -Wshadow --pedantic
ERRORS = -Wvla -Werror
GCC = gcc -std=c99 -g $(WARNINGS) $(ERRORS)
SRCS = main.c add.c
OBJS = $(SRCS:%.c=%.o)
addprog: $(OBJS)
	$(GCC) $(OBJS) -o addprog
test1 addprog
	cat inputs/input1 | $<
%.o: %.c
	$(GCC) -c $< -o $@

clean:
	rm $(OBJS) addprog
```