CS 314 Principles of Programming Languages

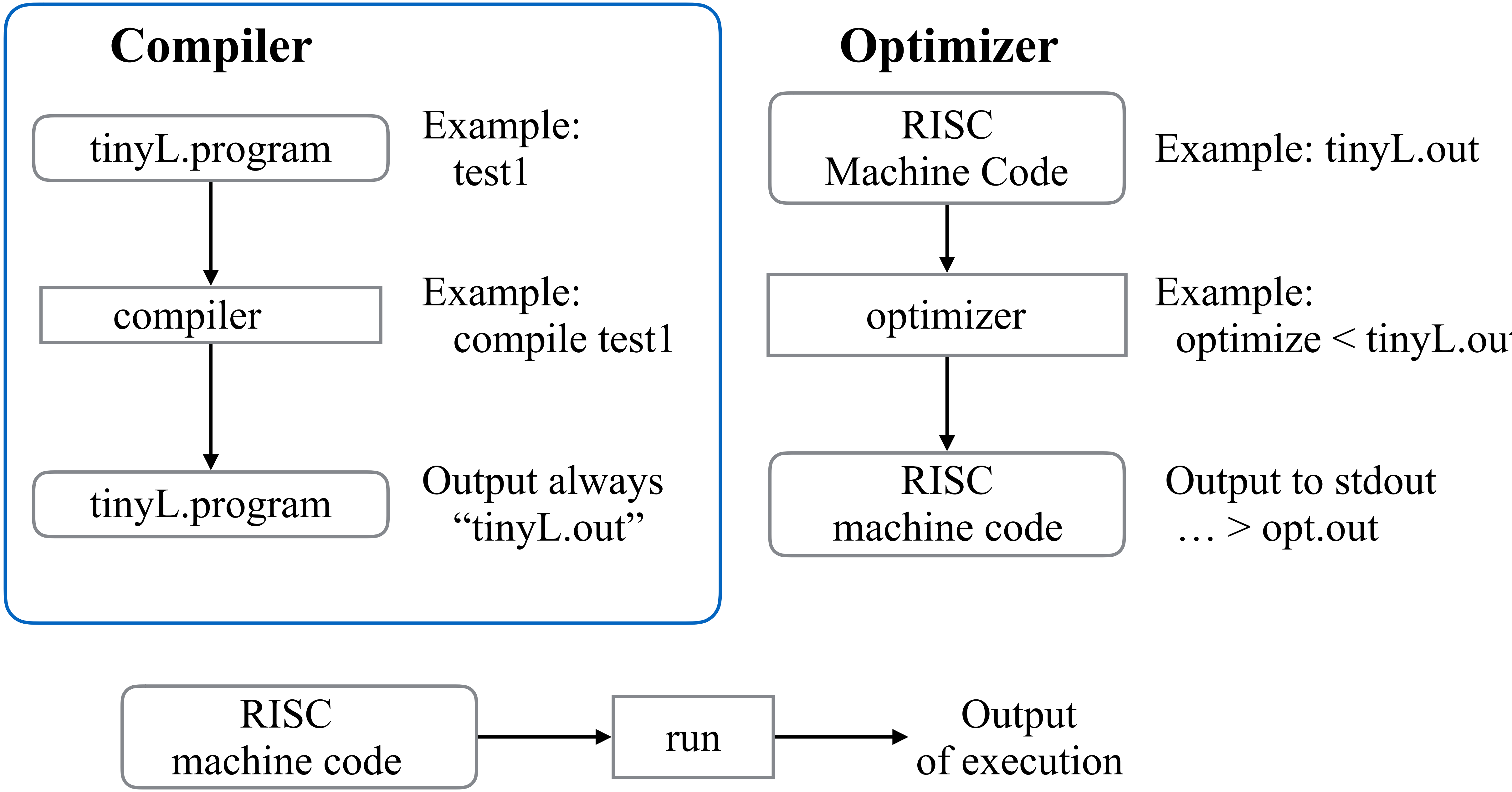
Lecture 11: Programming in C

Zheng (Eddy) Zhang
Rutgers University
February 19, 2018

Class Information

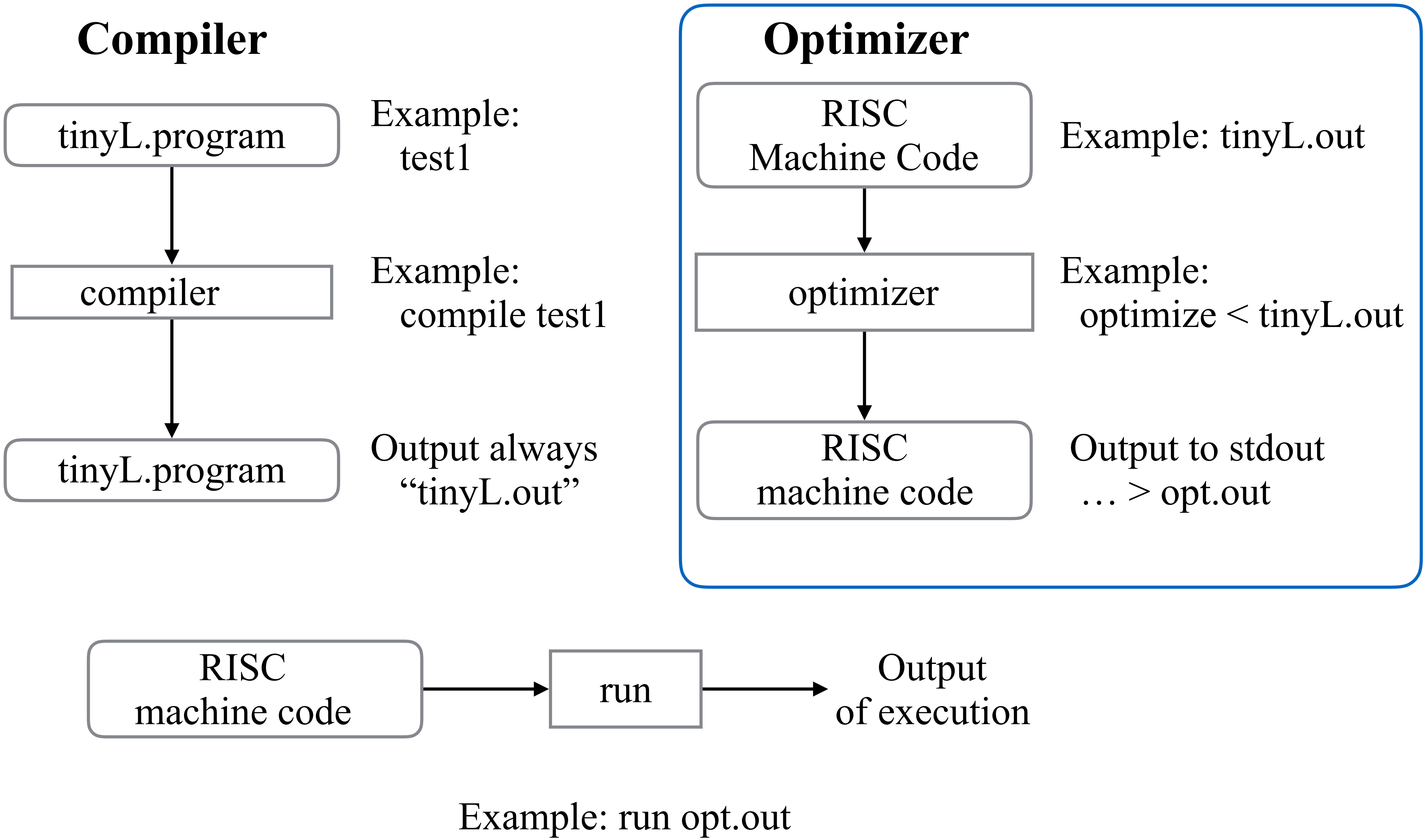
- Project 1 posted, due Tuesday, 3/6 11:55 pm EST.
- Homework 4 will be posted today.

Project 1: overview

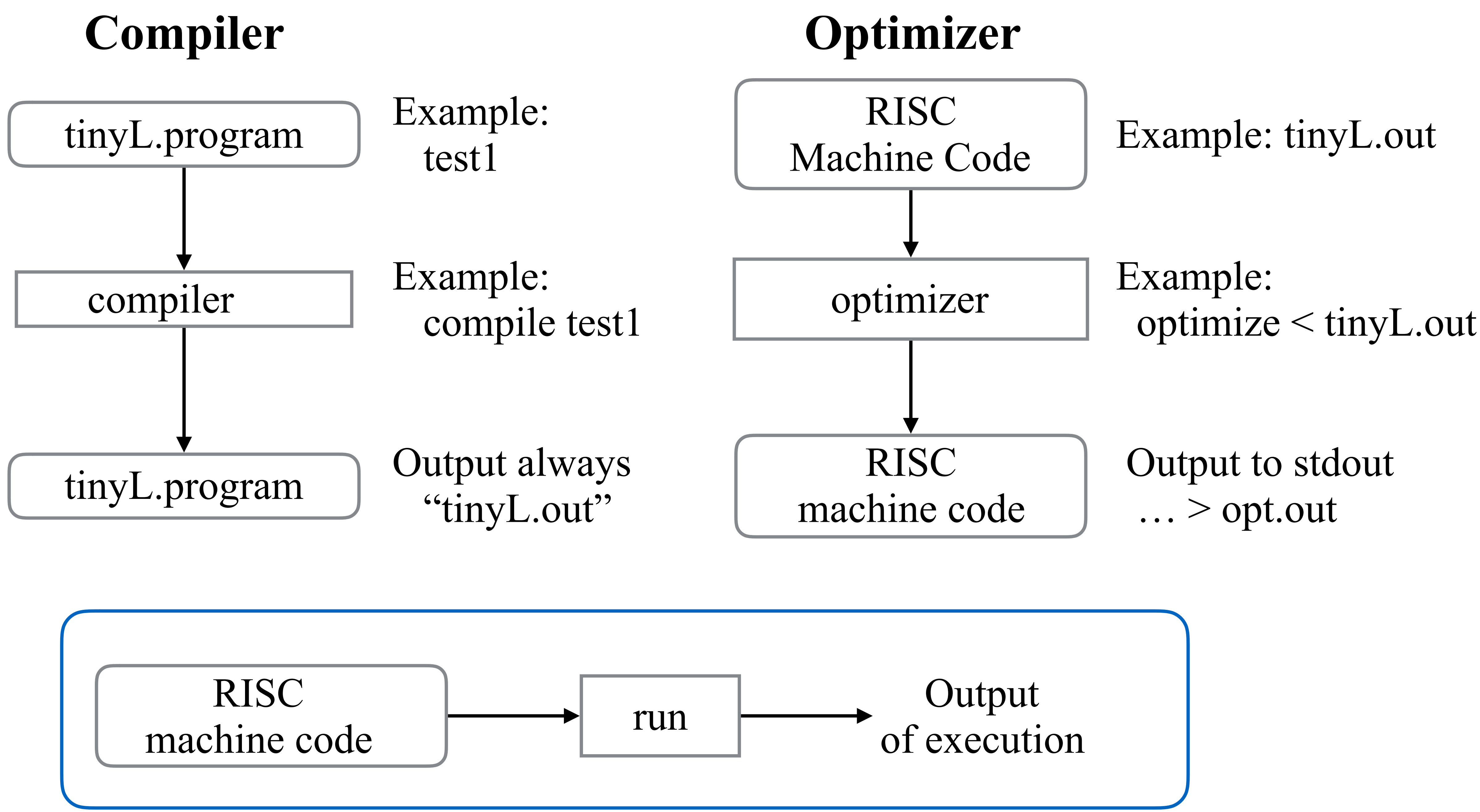


Example: run opt.out

Project 1: overview



Project 1: overview



Example: run opt.out

Project 1: Dead Code Elimination

Redundant code elimination: Eliminate code without changing the semantics of the program. If the execution of an operation or an instruction does not contribute to input/output behavior of the program, the instruction is considered as dead code and therefore can be eliminated.

Example:

Original Code

```
LOADI Rx #c1
LOADI Ry #c2
LOADI Rz #c3
ADD R1 Rx Ry
MUL R2 Rx Ry
STORE a R1
WRITE a
```

Optimized Code

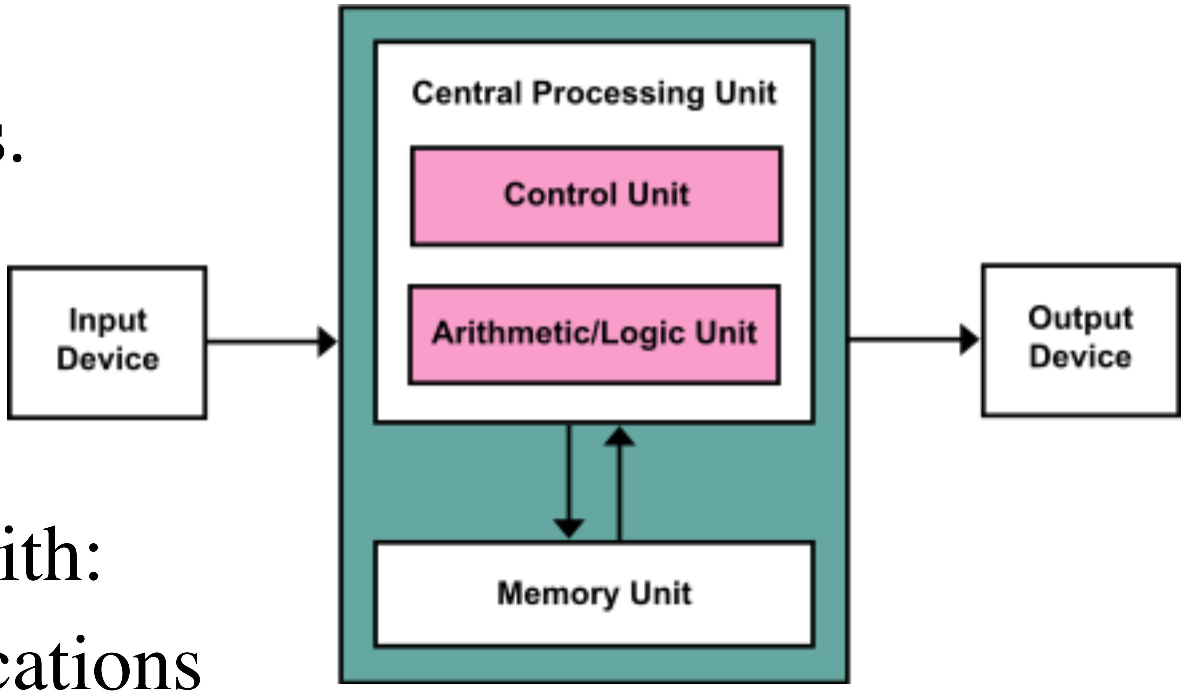
```
LOADI Rx #c1
LOADI Ry #c2
ADD R1 Rx Ry
STORE a R1
WRITE a
```

See project description for more details.

Review: Imperative Programming Language

Imperative:

Sequence of state-changing actions.



- Manipulate an abstract machine with:
  1. Variables naming memory locations
  2. Arithmetic and logical operations
  3. Reference, evaluate, assign operations
  4. Explicit control flow statements
- Key operations: Assignment and control flow
- Fits the Von Neumann architecture closely

C: An Imperative Programming Language

Expressions: include procedure and function calls and assignments, and thus can have side-effects

Control Structures:

- if statements, with or without else clauses
- loops, with break and continue exits

  while (< expr >) < stmt >

  do < stmt > while (< expr >)

  for (< expr >; < expr >; < expr >) < stmt >

- switch statements
- goto with labelled branch targets

Data Types in C

- Primitives: char, int, float, double
- Aggregates: arrays, structures

```
char a[10], b[2][10];
structure rectangle{
  structure point p1;
  structure point p2;
}
```

- Enumerations: collection of sequenced values
- Pointers

&i	address of i
*p	dereferenced value of p
p + 1	pointer arithmetic

```
int *p, i;
p = &i;
*p = *p + 1;
```

Basic Comparisons (Incomplete)

C	JAVA
Basic types: int, double, char, boolean	Primitive types: int, double, char, boolean
Pointer (to a value)	**Reference (to an object)**
Aggregates: array, **struct**	Aggregates: array, **object (class)**
Control Flow:if-else, switch, while, break, continue, for, return, **goto**	Control Flow:if-else, switch, while, break, continue, for, return
Logic Operators: ||, &&, !	Logic Operators: ||, &&, !
Logical Comparisons: ==, !=	Logical Comparisons: ==, !=
Numeric Comparisons: <, >, <=, >=	Numeric Comparisons: <, >, <=, >=
string as **char* array**	**String** as an object

Compile and Run a C Program

test.c:

```
#include <stdio.h>

int main (void)
{
        int x, y;
        printf("First number: \n"); scanf("%d", &x);
        printf("Second number: \n"); scanf("%d", &y);
        printf("%d + %d = %d \n", x, y, x + y);
        printf("%d - %d = %d \n", x, y, x - y);
        printf("%d * %d = %d \n", x, y, x * y);
        return 0;
}
```

gcc test.c:	calls GUN C compiler, and generate executable a.out
./a.out:	runs the executable
gcc -o run test.c:	compiles program and generate executable run
gcc -g test.c:	generate executable a.out with debugging info
gdb a.out:	run debugger on a.out; online documentation *man gdb*

Compile and Run a C Program

```
> gcc test.c
> ./a.out

First number:
4
Second number:
12
4 + 12 = 16
4 - 12 = -8
4 * 12 = 48
>
```

START PROGRAMMING IN C NOW!

Debugging C Programs

```
console%  gdb a.out
(gdb)  list
1          #include <stdio.h>
2          int main (void)
3          {
4              int x, y;
5              printf(“First number: \n“);  scanf(“%d”, &x);
6              printf(“Second number: \n“);  scanf(“%d”, &y);
7              printf(“%d + %d = %d \n", x, y, x + y);
(gdb)  break 7
```

Debugging C Programs

```
console%  gdb a.out
(gdb)  list
1           #include <stdio.h>
2           int main (void)
3           {
4               int x, y;
5               printf(“First number: \n“);  scanf(“%d”, &x);
6               printf(“Second number: \n“);  scanf(“%d”, &y);
7               printf(“%d + %d = %d \n", x, y, x + y);
(gdb)  break 7
Breakpoint 1 at 0x1052c: file test.c, line 7.
```

Debugging C Programs

```
console%  gdb a.out
(gdb)  list
1        #include <stdio.h>
2        int main (void)
3        {
4            int x, y;
5            printf(“First number: \n“);  scanf(“%d”, &x);
6            printf(“Second number: \n“);  scanf(“%d”, &y);
7            printf(“%d + %d = %d \n", x, y, x + y);
(gdb)  break 7
Breakpoint 1 at 0x1052c: file test.c, line 7.
(gdb)  run
```

Debugging C Programs

```
console%  gdb a.out
(gdb)  list
1          #include <stdio.h>
2          int main (void)
3          {
4              int x, y;
5              printf(“First number: \n“);  scanf(“%d”, &x);
6              printf(“Second number: \n“);  scanf(“%d”, &y);
7              printf(“%d + %d = %d \n", x, y, x + y);
(gdb)  break 7
Breakpoint 1 at 0x1052c: file test.c, line 7.
(gdb)  run
Starting program: /…/a.out
First number:
4
Second number:
12
```

Debugging C Programs (contd.)

```
Breakpoint 1, main ( ) at test.c: 7
7                       printf("%d + %d = %d \n", x, y, x + y);
(gdb) print x
$1 = 4
(gdb) print y
$2 = 12
(gdb) cont
Continuing.
4 + 12 = 16
4 - 12 = -8
4 * 12 = 48
Program exited normally.
(gdb) quit
```

Debugging C Programs (contd.)

```
Breakpoint 1, main ( ) at test.c: 7
7                   printf("%d + %d = %d \n", x, y, x + y);
(gdb) print x
$1 = 4
(gdb) print y
$2 = 12
(gdb) cont
Continuing.
4 + 12 = 16
4 - 12 = -8
4 * 12 = 48
Program exited normally.
(gdb) quit
```

Debugging C Programs (contd.)

```
Breakpoint 1, main ( ) at test.c: 7
7                    printf("%d + %d = %d \n", x, y, x + y);
(gdb) print x
$1 = 4
(gdb) print y
$2 = 12
(gdb) cont
Continuing.
4 + 12 = 16
4 - 12 = -8
4 * 12 = 48
Program exited normally.
(gdb) quit
```

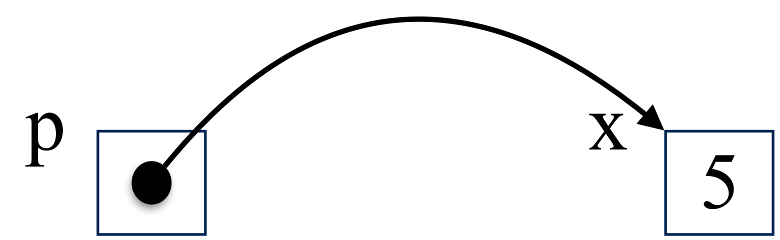
Pointers in C

Pointer: Variable whose R-value (content) is the L-value (address) of a variable

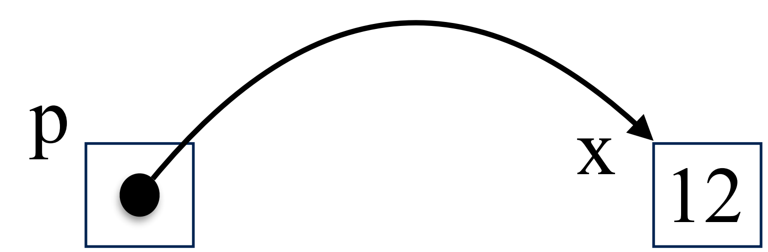
- “address-of” operator &
- deference (“content-of”) operator *

int *p, x;

p = &x;
*p = 5;

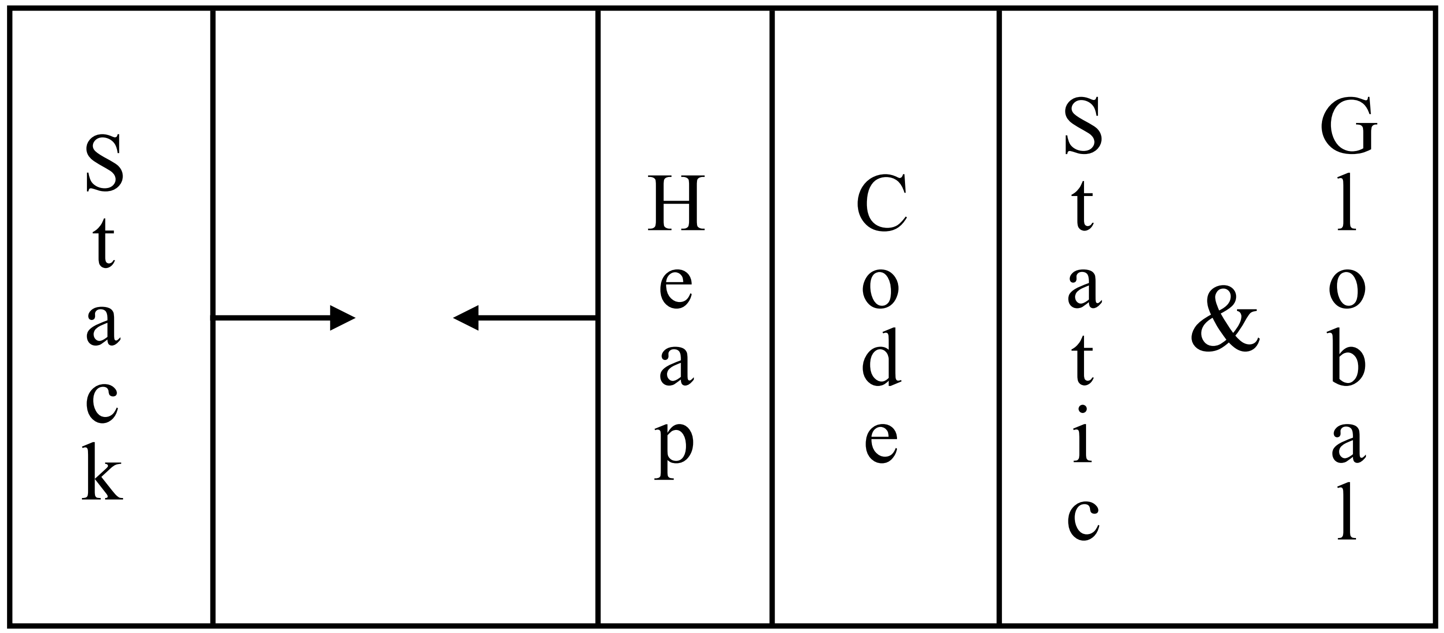


x = 12;



Review: Run-time Storage Organization

Typical memory layout



Most Language runtime layout the address space in a similar way

- Pieces (stack, heap, code & globals) may move, but all will be there
- Stack and heap grow toward each other
- Arrays live on one of the stacks, in the global area, or in the heap

Review: Stack vs Heap

Stack:

- Procedure activations, statically allocated local variables, parameter values
- Lifetime same as subroutine in which variables are declared
- Stack frame is pushed with each invocation of a subroutine, and popped after subroutine exit

Heap:

- Dynamically allocated data structure, whose size may not be known in advance
- Lifetime extends beyond subroutine in which they are created
- Must be explicitly freed or garbage collected

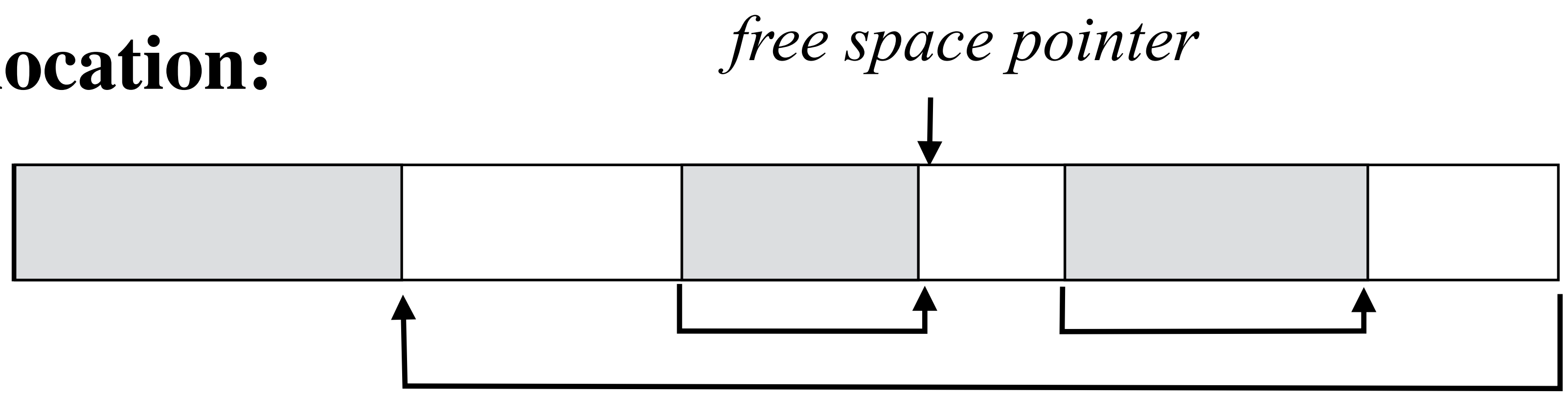
Example: Maintaining Free Lists

- **Allocate**: continuous block of memory; remove space from free list (here: singly-linked list).

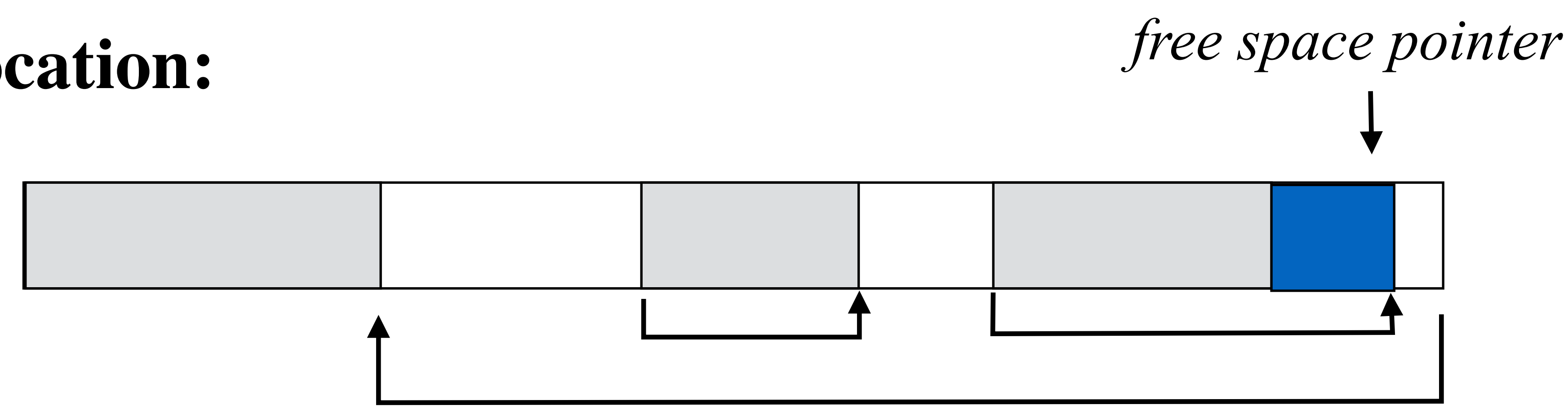
- **Free**: return to free list after coalescing with adjacent free storage (if possible); may initiate compaction.

Example: Maintaining Free Lists

Before **Allocation:**



After **Allocation:**

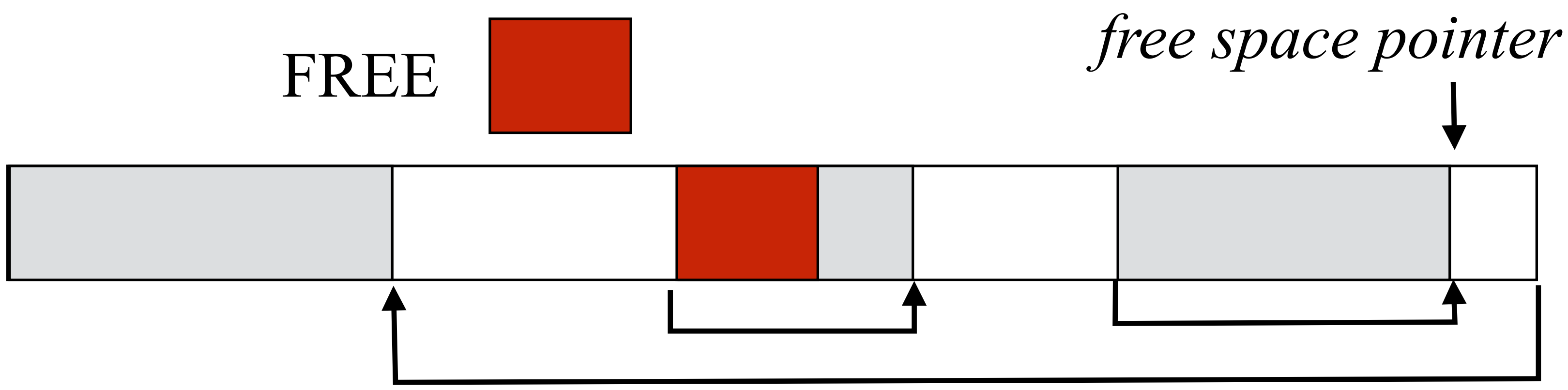


ALLOCATED

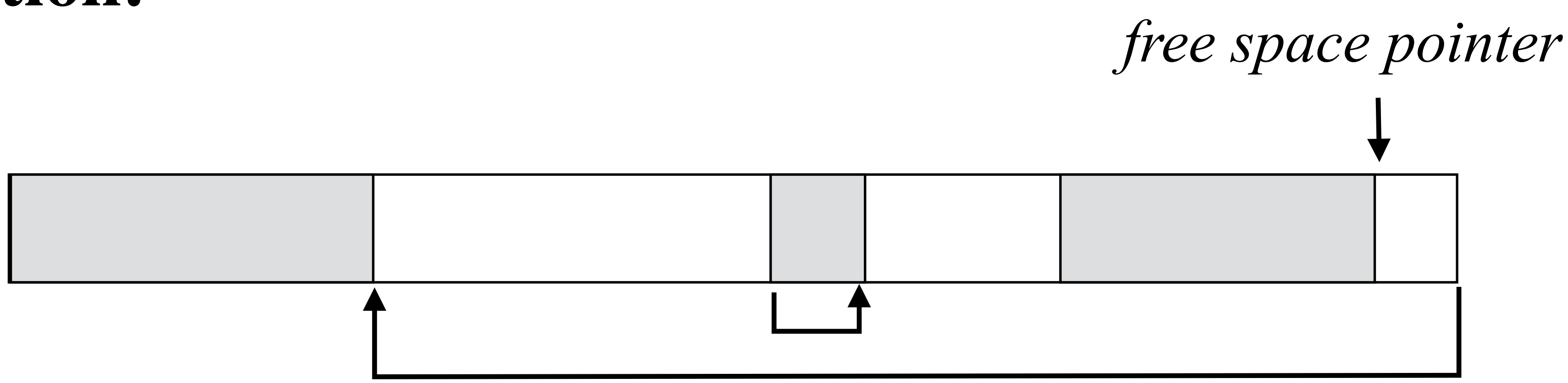
NEWLY ALLOCATED

Example: Maintaining Free Lists

Before De-allocation:



After **De-allocation:**



Potential Issues with Explicit Control of Heaps

- Dangling references
  - Storage pointed to is freed, but pointer is not set to NULL
  - Able to access storage whose values are not meaningful
- Garbage
  - Objects in heap that cannot be accessed by the program any more
  - Example

```
int *x, *y;
x = (int*) malloc (sizeof (int));
y = (int*) malloc (sizeof (int));
x = y;
```

- Memory leaks
  - Failure to release (reclaim) memory storage build up overtime

Potential Issues with Explicit Control of Heaps

- Dangling references
  - Storage pointed to is freed, but pointer is not set to NULL
  - Able to access storage whose values are not meaningful
- Garbage
  - Objects in heap that cannot be accessed by the program any more
  - Example

```
int *x, *y;
x = (int*) malloc (sizeof (int));
y = (int*) malloc (sizeof (int));
x = y;
```

- Memory leaks
  - Failure to release (reclaim) memory storage build up overtime

Potential Issues with Explicit Control of Heaps

- Dangling references
  - Storage pointed to is freed, but pointer is not set to NULL
  - Able to access storage whose values are not meaningful
- Garbage
  - Objects in heap that cannot be accessed by the program any more
  - Example

```
int *x, *y;
x = (int*) malloc (sizeof (int));
y = (int*) malloc (sizeof (int));
x = y;
```

- Memory leaks
  - Failure to release (reclaim) memory storage build up overtime

Example: Singly-Linked List

```
#include <stdio.h>
#include <stdlib.h>
/* TYPE DEFINITION */
typedef struct cell listcell;

struct cell
{       int num;
        listcell *next;
};

/* GLOBAL VARIABLES */
listcell  *head, *new_cell, *current_cell;
```

Example: Singly-Linked List

Let's deallocate,i.e., free all list elements

```
#include <list.h>
/* GLOBAL VARIABLES */
listcell *head, *new_cell, *current_cell;
int main(void){
        /* CREATE FIRST LIST ELEMENT */
        ...

        /* CREATE NINE MORE ELEMENTS */
        ...

        /* DEALLOCATE LIST */
        for(current_cell = head;
            current_cell != null;
            current_cell = current_cell -> next){
                free(current_cell);
        }
        ...
}
```

Does this work?

Example: Singly-Linked List

Let’s deallocate,i.e., free all list elements

```
#include <list.h>
/* GLOBAL VARIABLES */
listcell *head, *new_cell, *current_cell;
int main(void){
        /* CREATE FIRST LIST ELEMENT */
        ...

        /* CREATE NINE MORE ELEMENTS */
        ...

        /* DEALLOCATE LIST */
        for(current_cell = head;
            current_cell != null;
            current_cell = current_cell -> next){
                free(current_cell);
        }
        ...
}
```

What went wrong?

Uninitialized variables and “dangerous” casting

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

int main(void){
        int *a;

        *a = 12;
        printf("%x,%x: %d\n", &a, a, *a);

        a = (int *)12;
        printf("%d \n", *a);
}
```

```
> a.out
effff60c effff68c: 12
segmentation fault (core dumped)
```

Note: Segmentation faults result in the generation of a core file which can be rather large. Don’t forget to delete it.

What went wrong?

That's better!

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
        int *a = NULL; /* good practice */

        a = (int *)malloc(sizeof(int));

        *a = 12;
        printf("%x,%x: %d\n", &a, a, *a);
}
```

```
> a.out
effff60c 20900: 12
```

What went wrong?

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

int main(void){
        int i;

        char* string = "Hello, how are you today.";
        printf("\n%s\n", string);

        for(i = 0; string[i] != '.'; i++){
                if(string[i] = ' ')
                        for(; string[i] = ' '; i++);
                printf("%c", string[i]);
        }
        printf(".\n");
}
```

```
> a.out
Hello, how are you today.
Segmentation fault (core dumped)
```

What went wrong?

“ = ” is not the same as “ == ”

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

int main(void){
	int i;

	char* string = "Hello, how are you today.";
	printf("\n%s\n", string);

	for(i = 0; string[i] != '.'; i++){
		if(string[i] == ' ')
			for(; string[i] == ' '; i++);
		printf("%c", string[i]);
	}
	printf(".\n");
}
```

```
> a.out
Hello, how are you today.
Hello,howareyoutoday.
```

What went wrong?

“ Aliasing ” and freeing memory

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

int main(void){
        int *a = NULL;
        int *b = NULL;
        int *c = NULL;

        a = (int *)malloc(sizeof(int));
        b = a;
        *a = 12;
        printf("%x %x: %d\n", &a, a, *a);
        printf("%x %x: %d\n", &b, b, *b);
        free(a);
        printf("%x %x: %d\n", &b, b, *b);

        c = (int *)malloc(sizeof(int));
        *c = 10;
        printf("%x %x: %d\n", &c, c, *c);
        printf("%x %x: %d\n", &b, b, *b);
}
```

```
> a.out
effff60c 209d0: 12
effff608 209d0: 12
effff608 209d0: 12
effff604 209d0: 10
effff604 209d0: 10
```

What went wrong?

Use a subroutine to create an object

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

/* TYPE DEFINITION */
typedef struct cell listcell;
struct cell
{
        int num;
        listcell *next;
};
listcell *head = NULL;
listcell *create_listcell(){
        listcell new;
        new.num = -1;
        new.next = NULL;
        return &new;
}
int main(void){
        head = create_listcell();
        printf("head -> num = %d\n", head -> num);
}
```

What went wrong?

Use a subroutine to create an object (cont.)

```
> gcc stack.c
  stack.c: In function "create_listcell":
  stack.c:17: warning: function returns address of local variable

> ./a.out

head —> num = -1
```

What went wrong?

Use a subroutine to create an object: malloc

```
#include <stdio.h>
#include <stdlib.h>
/* TYPE DEFINITION */
typedef struct cell listcell;
struct cell
{
	int num;
	listcell *next;
};
listcell *head = NULL;
listcell *create_listcell(){
	listcell *new;
	new = (listcell *)malloc(sizeof(listcell));
	new -> num = -1;
	new -> next = NULL;
	return new;
}
int main(void){
	head = create_listcell();
	printf("head -> num = %d\n", head -> num);
}
```

What went wrong?

Use a subroutine to create an object: malloc (cont.)

> gcc heap.c
> ./a.out

head —> num = -1

Pointers and Arrays in C

Pointers and arrays are similar in C:

- Array name is a pointer to a[0]:

```
int a[10];
int *pa;
pa = &a[0];
```

pa and a have the same semantics

- Pointer arithmetic is array indexing

pa + 1 and a + 1 point to a[1]

- Exception: an array name is a constant pointer

a++ is ILLEGAL
a = pa is ILLEGAL (pa = a is LEGAL!)

Next Lecture

Things to do:

- Start programming in C.
- Read Scott, Chapter 8.1 - 8.2; ALSU 7.1 - 7.3.
- Next time:
  - Procedure abstractions; Runtime stack; Scoping