CS1020E Tutorial + Lab 01

Mark NG

a0116298@u.nus.edu
http://mollymr305.github.io

August 26, 2016

Introduction

About myself.

1. Final year undergraduate.
2. Majors: Mathematics, Statistics.
3. Minor: Computer Science.
4. CS Modules:
   - Finished CS1010S, CS1020, CS2010, CS2100, CS2102, and CS3233.
   - Currently reading CS3244.
5. Will there be slides for every tutorial/lab session? It depends...

Introduction

1. Welcome to Lab + Tutorial Group 07!
2. Discussion of weekly tutorial solutions.
3. *Some* discussion of take home lab solutions.
4. Ask questions if you are not sure! I'll try to help.

Introduction

Official tutorial solutions will be posted at:

`http://www.comp.nus.edu.sg/~stevenha/cs1020e.html`

These slides are written by myself (disclaimer!)

Question 1

What is the output, and why?

```
#include <iostream>
using namespace std;

int main () {
  int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
  int h = f-- && e++ && d++ && c-- || b++ || a++;
  if (g = 9) {
    cout << a <<  b << c << d << e << f << g << h << endl;
  } else {
    cout << h << g << f << e << d << c << b << a << endl;
  }
  return 0;
}
```

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (−−), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(((f-- && e++) && d++) && c--) || b++ || a++
```

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(((f-- && e++) && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(((f-- && e++) && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
true	-1	1	1	0	2	2	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(((true && e++) && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	0	2	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(((true && e++) && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
true	-1	1	1	0	2	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(((true && true) && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	0	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(((true) && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	0	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
((true && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	0	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
((true && d++) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
false	-1	1	1	0	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
((true && false) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
((false) && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(false && c--) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

`(false && c--) || b++ || a++` $\Longleftarrow$ short-circuiting (why?)

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(false) || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

`false || b++ || a++`

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	1	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
false || b++ || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
true	-1	1	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
false || true || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	2	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
(false || true) || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	2	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

```
true || a++
```

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	2	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

`true` || a++ $\Longleftarrow$ short-circuiting again (why?)

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	2	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

true

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	2	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

Therefore, the final value of `h` is 1 (why?).

Variable	a	b	c	d	e	f	g
Initial Value	-1	1	1	0	2	2	0
	-1	2	1	1	3	1	0

Question 1

Left-to-right. The expression is evaluated from left to right.
Operator precedence. Suffix increment (++) and decrement (--), followed by logical AND (&&), and finally logical OR (||).
Short-circuiting. Due to logical operators, part of the statement might be skipped if the logical result is certain.

```
int a = -1, b = 1, c = 1, d = 0, e = 2, f = 2, g = 0;
int h = f-- && e++ && d++ && c-- || b++ || a++;
```

Therefore, the final value of `h` is 1 (why?).

Variable	a	b	c	d	e	f	g	h
Initial Value	-1	1	1	0	2	2	0	
Final Value	-1	2	1	1	3	1	0	1

Question 1

Last part of Q1. The difference between `g = 9` and `g == 9`!

```
if (g = 9) {
  cout << a <<  b << c << d << e << f << g << h << endl;
} else {
  cout << h << g << f << e << d << c << b << a << endl;
}
```

Variable	a	b	c	d	e	f	g	h
Initial Value	-1	1	1	0	2	2	0	
Final Value	-1	2	1	1	3	1	0	1

Question 1

Last part of Q1. The difference between `g = 9` and `g == 9`!

```
if (g = 9) {
  cout << a <<  b << c << d << e << f << g << h << endl;
} else {
  cout << h << g << f << e << d << c << b << a << endl;
}
```

Variable	a	b	c	d	e	f	g	h
Initial Value	-1	1	1	0	2	2	0	
Final Value	-1	2	1	1	3	1	9	1

What happens? The `if` block is executed. Note that `g = 9` assigns the value of 9 to g, then uses this value.

Question 1

Last part of Q1. The difference between `g = 9` and `g == 9`!

```
if (g = 9) {
  cout << a <<  b << c << d << e << f << g << h << endl;
} else {
  cout << h << g << f << e << d << c << b << a << endl;
}
```

Variable	a	b	c	d	e	f	g	h
Initial Value	-1	1	1	0	2	2	0	
Final Value	-1	2	1	1	3	1	9	1

Output: -12113191

Question 1

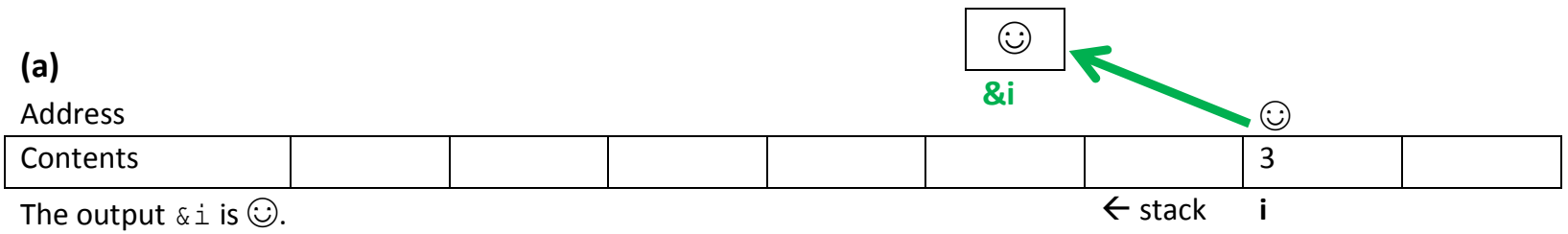
Moral of the story: Always write clean, readable code. Give appropriate variable names. Whatever convention you use, try to keep it consistent.

Question 2

Solutions here are taken directly from `T1_ans.pdf`.

Part (a).

```
int i = 3;
cout << &i;
```



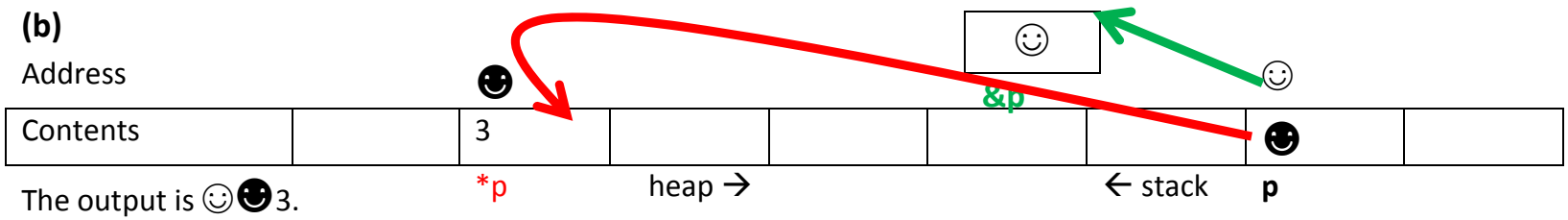
The **ampersand** `&` in the second line is the **address-of operator**. It is used to read the memory address the variable is located in. Therefore, the result of `&i` is ☺.

Question 2

Solutions here are taken directly from `T1_ans.pdf`.

Part (b).

```
int* p = new int(3);
cout << &p << p << *p;
```



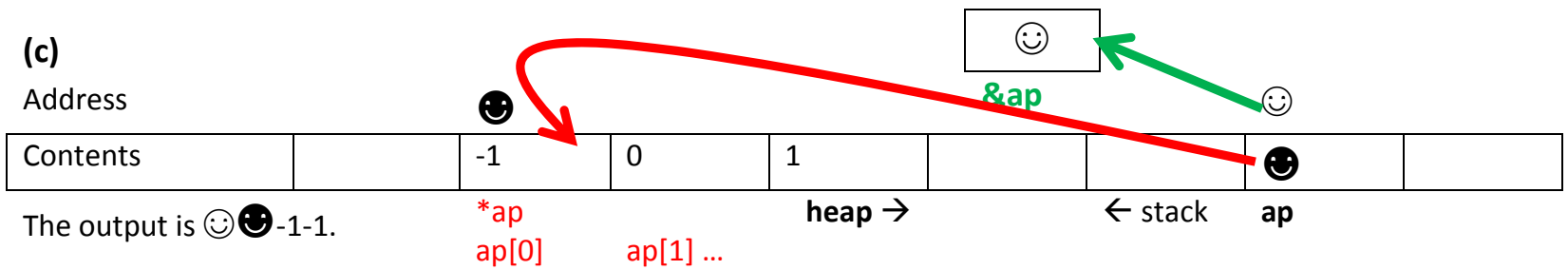
The asterisk ***** in the first line is part of the datatype, i.e. `p` is an "integer pointer" variable.
The **asterisk in the second line is the indirection** (dereferencing) **operator**. It is used to read the contents of the memory address pointed to by `p`, which has the value of 3.

Question 2

Solutions here are taken directly from `T1_ans.pdf`.

Part (c).

```
int* ap = new int[3];
for (int i = 0; i < 3; i++)
  ap[i] = i - 1;
cout << &ap << ap << *ap << ap[0];
```



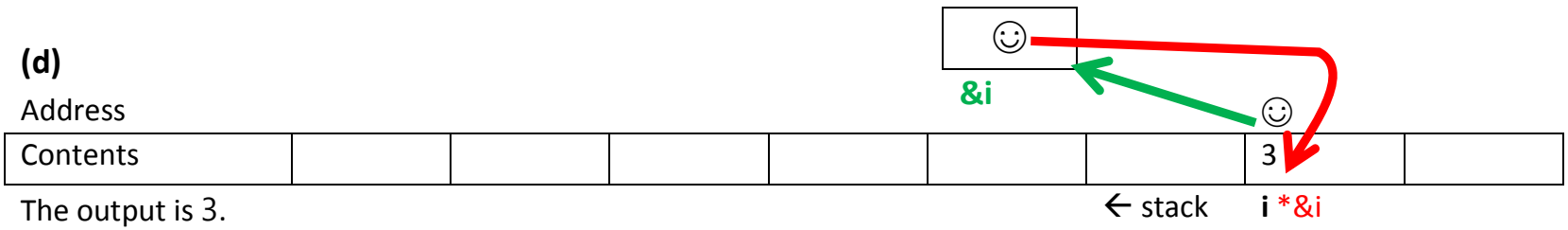
The first line creates an integer array of 3 elements on the **heap**, and assigns the address to an "integer pointer" variable. `*ap` (indirection) and `ap[0]` (array subscript) are equivalent, both dereferencing operators. They are both equivalent to `*(ap + 0)`.

Question 2

Solutions here are taken directly from `T1_ans.pdf`.

Part (d).

```
int i = 3;
cout << *&i;
```



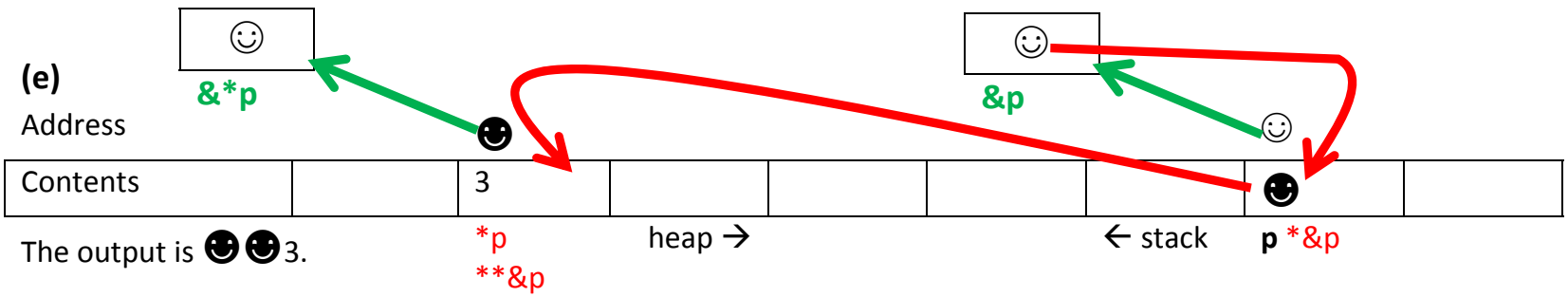
The * and & (indirection and address-of) operators have right-to-left associativity. This means that in *&i, the address of i is first taken (&i), following which, the contents of the memory address pointed to by &i are read.

Question 2

Solutions here are taken directly from `T1_ans.pdf`.

Part (e).

```
int* p = new int(3);
cout << *&p << &*p << **&p;
```



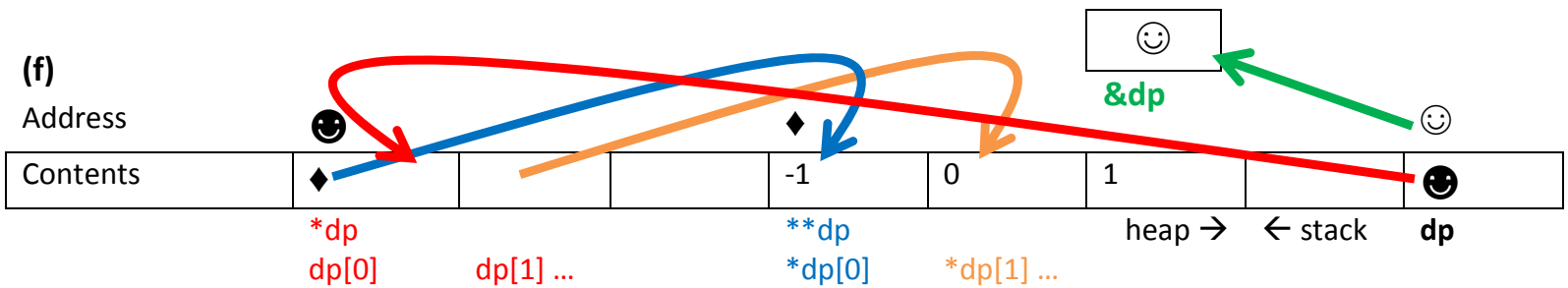
Remember, what is printed out is always the contents of some memory location. The contents may be a value (as when printing `**&p`), or another memory location (as when printing `*&p`).

Question 2

Solutions here are taken directly from `T1_ans.pdf`.

Part (f).

```
int** dp = new int* [3];
for (int i = 0; i < 3; i++)
  dp[i] = new int(i-1);
cout << &dp << dp << *dp << dp[0] << **dp << *dp[0];
```



The output is ☺☻♦♦-1-1.

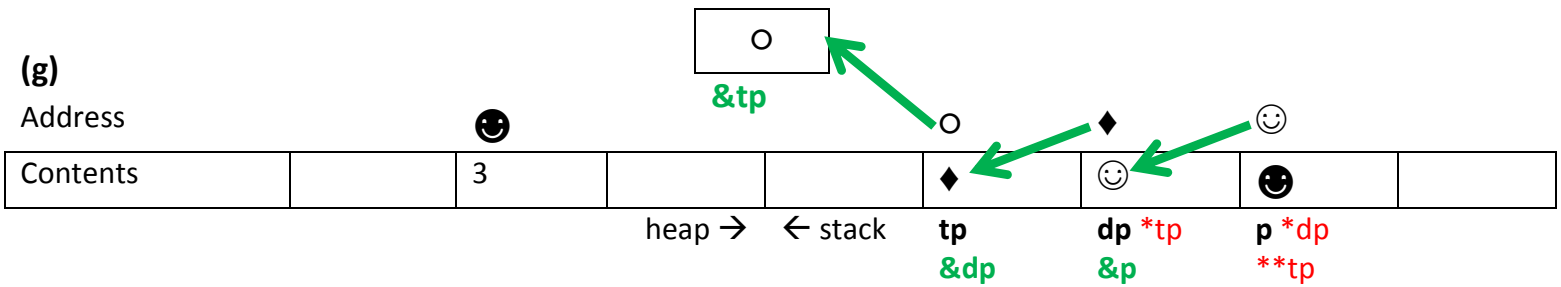
new `int*` **[**`3`**]** creates an array of 3 elements. Each element is an "integer pointer", i.e. each element contains the memory address of a location that stores an int. **[]** has a higher precedence than *****, so *****`dp`**[**`0`**]** moves to the 0th element of the array, and then to the location of the integer containing -1.

Question 2

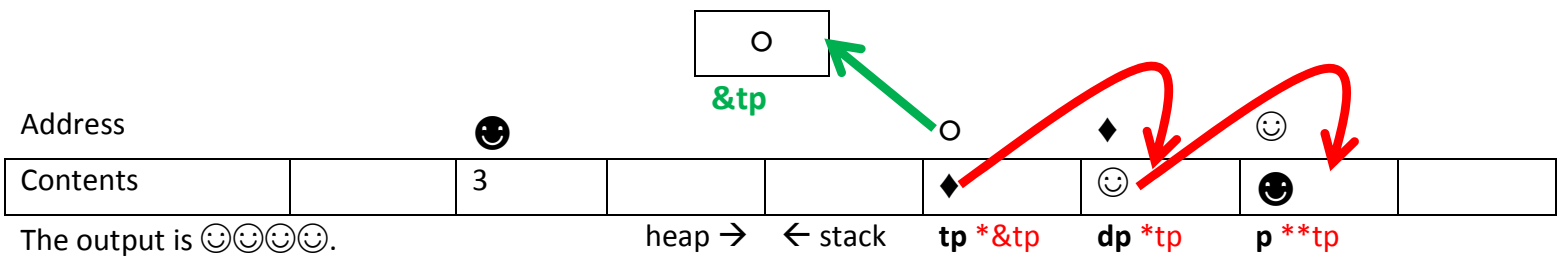
Solutions here are taken directly from `T1_ans.pdf`.

Part (g).

```
int* p = new int(3);
int** dp = &p;
int*** tp = &dp;
cout << *tp << &**tp << *&*tp << **&tp;
```



By now, you may have observed that `&` and **`*`** are somewhat the opposite of each other.



The output is ☺☺☺☺.

Question 3

Open discussion. Official solutions will be uploaded either way.

Question 3

Skeleton code.

```
class Animal {
    /* insert code here */
};
class Song {
  private:
    Animal** _animals;
    const int _size;
  public:
    Song() { /* insert code here */ }
    ~Song() { /* insert code here */ }
    void display() {
      for (int i = 0; i < _size; i++)
        cout << endl; /* insert code here */
    }
};
int main() {
  Song song;
  song.display();
  return 0;
}
```

Let's take a short break!

Exercise 1

This should be quite simple...

Exercise 2

- **Ordinary.** You need to simulate the problem description. Be careful of corner cases!
- **First Improvement.** Make some optimizations, i.e. reduce the total number of operations needed.
- **Second Improvement.** Not in syllabus. For more information: take CS2010, CS3233, and/or read Competitive Programming 3 (book).

Kattis Online Judge

Create an account at:

`https://open.kattis.com/`

Kattis Online Judge

Today we will solve two problems LIVE:

1. `https://open.kattis.com/problems/hello`
2. `https://open.kattis.com/problems/pizza2`

Any Questions?

See you next week!