CS1020E: Data Structures and Algorithms I

Tutorial 1 – Basic C++, OOP Problem Solving

(Week 3, starting 22 August 2016)

1. *Evaluation Order (Note: You can use any other C++ code editor/compiler).*

Examine the code snippet. What is the **output**, and **why**?

Tip: Check your answer! Create a program in vim, paste main method within. Compile and run in sunfire.

```
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;
}
```

-= WarmUp =-

a	b	c	d	e	f	g	h

Related Concepts

- Operator precedence vs evaluation order
- Conditions, logical operator short-circuiting
- Assignment vs equality

Answer

```
-12113191
```

The most important lesson here is to **write readable code**! Write code first for humans to read, then for the computer to understand. Complex **conditions** and **poor variable names** make life difficult for humans. Separate complex conditions into simpler statements, using descriptive variable names.

C++'s operator precedence can be found at

http://en.cppreference.com/w/cpp/language/operator_precedence

++ and -- here have highest **precedence**, followed by &&, then ||, and finally the assignment operator =. However, this does NOT mean that ALL ++ and -- in a statement are executed first. Rather, the **expression is evaluated left to right in general**, taking into account operator precedence.

Because of logical operators && and ||, **short-circuiting** may take place, i.e. **part of a statement may be skipped** because the preceding part of the statement already determines the result. If no short-circuiting takes place, the expression (before assignment to h) is evaluated as:

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

A condition is **logically true if** an expression evaluates to `true` (bool) or a non-zero value, false otherwise. `f--` (**post-decrement**) is not to be confused with `--f` (pre-decrement). `f--` means: use the current value of f in evaluating the expression, then decrease the value of f in memory by 1. Pre-decrement works the other way round.

-1	1	1	0	2	2	1	?
a	b	c	d	e	f	g	expr

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

`f`, holding the value of 2, evaluates to `true`. It is then decremented. No short-circuiting occurs because the result of the expression within the innermost brackets is not known yet.

-1	1	1	0	2	1	0	?
a	b	c	d	e	f	g	expr

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

`e`, holding the value of 2, evaluates to `true`. It is then incremented.

-1	1	1	0	3	1	0	?
a	b	c	d	e	f	g	expr

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

`d`, holding the value of 0, evaluates to `false`. It is then incremented.

-1	1	1	1	3	1	0	?
a	b	c	d	e	f	g	expr

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

It is certain that the brackets now evaluate to `false`, so `c--` is **NOT executed** (short-circuiting).

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

`b`, holding the value of 1, evaluates to `true`. It is then incremented.

-1	2	1	1	3	1	0	?
a	b	c	d	e	f	g	expr

`true || a++`

It is certain that the expression now evaluate to `true`, so `a++` is **NOT executed** (short-circuiting).

-1	2	1	1	3	1	0	true
a	b	c	d	e	f	g	expr

`true` is **converted to** an `int` value of 1 and assigned to h.

Next, if the condition were `(g == 9)`, the **`else`** block would be executed. However, `(g = 9)` **assigns** the value of 9 to `g`, and **uses the new value** of `g` as the result of the expression. As `g` is now non-zero, the **`if`** block is executed.

Lesson: Avoid using the wrong operator in your code! Differentiate between = and ==.

In subsequent tutorials, you are expected to **trace through code** by yourself, as in Q1. Draw diagrams to help you understand **what happens in memory**, as in Q2.

2. *Understanding Pointers*

In OOP languages, pointers (so named in C++; may be called “references” in other languages) are widely used to locate one object from another. It is necessary to have a firm understanding of them.

For each of these cases, *independent of one another*, **draw** how the variables may appear in memory, and what **output** you would expect. You do not need to worry about the exact memory addresses, but you should find out how different expressions are related to each other. ☺, ☻, ○, and ♦ represent memory addresses.

(a) has been done for you. (d) to (g) may be more difficult. Remember to check your answer.

(a)
```
int i = 3;
cout << &i;
```

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

(c)
```
int* ap = new int[3];

for (int i = 0; i < 3; i++)
   ap[i] = i - 1;

cout << &ap << ap << *ap << ap[0];
```

(d)
```
int i = 3;
cout << *&i;
```

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

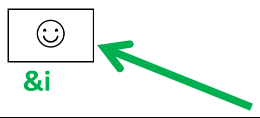
(g)
```
int* p = new int(3);
int** dp = &p;
int*** tp = &dp;

cout << *tp << &**tp <<
   *&*tp << **&tp;
```

(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];
```



(a)

Address							☺	
Contents							3	
The output &i is ☺.						← stack	i	

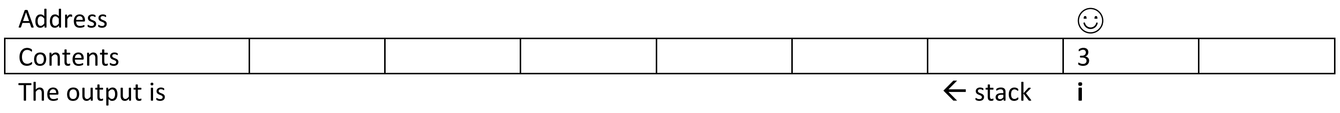
(b)

Address							☺	
Contents							☻	
The output is			heap →			← stack	p	

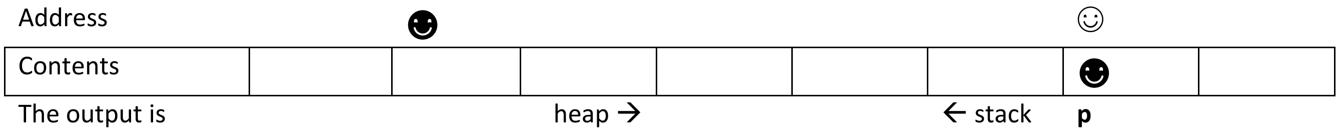
(c)

Address		☻					☺	
Contents							☻	
The output is				**heap →**		← stack	**ap**	

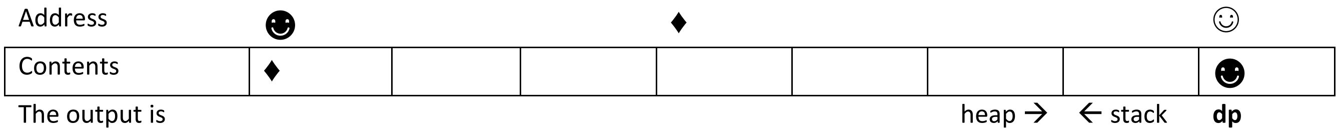
(d)



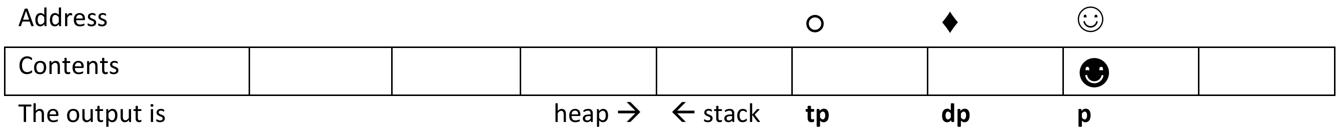
(e)



(f)



(g)



Also, when the **new** keyword is used, the system dynamically allocates memory for the newly created object. Therefore, the object ends up in a portion of memory called the **heap**. Otherwise, when **new** is not used, objects assigned to variables declared within functions reside on the **stack**.

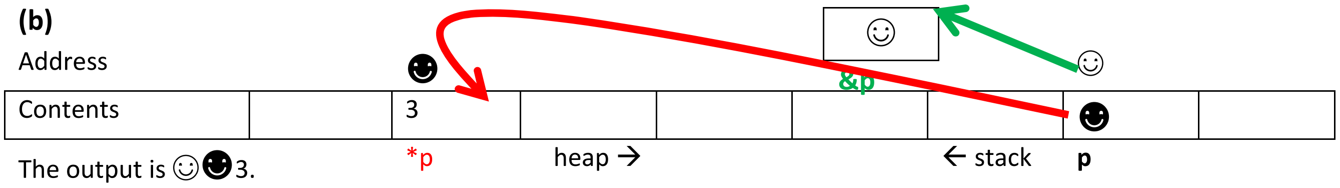
Why do we need to bother about heap vs stack memory? What other keyword and syntax is/are involved?

Answer

(a)

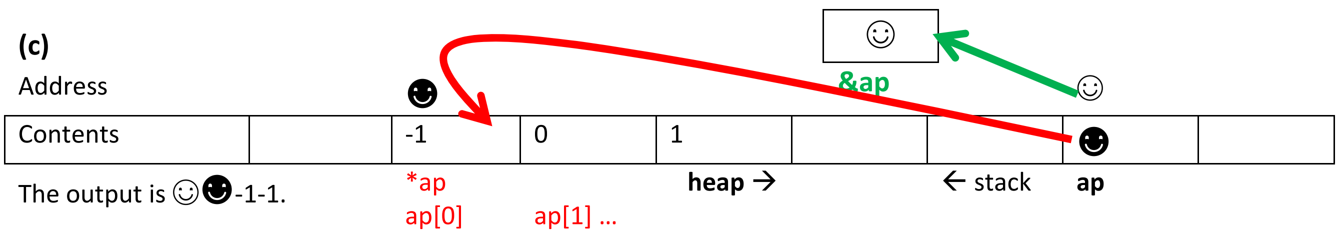
The **ampersand** & in the second line is the **address-of operator**. It is used to read the memory address the variable is located in. Therefore, the result of `&i` is ☺.

(b)



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.

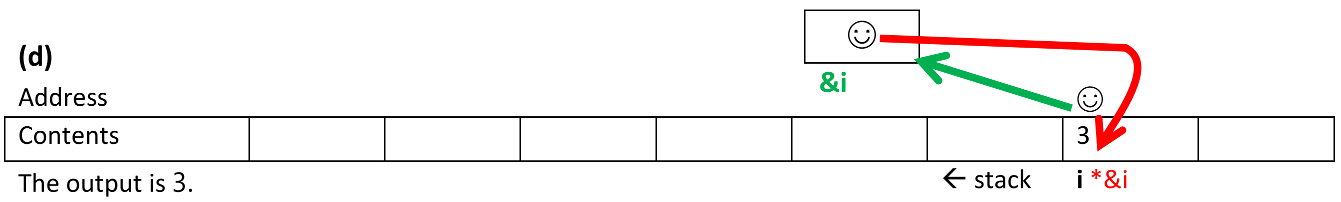
(c)



The output is ☺☻-1-1.

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)`.

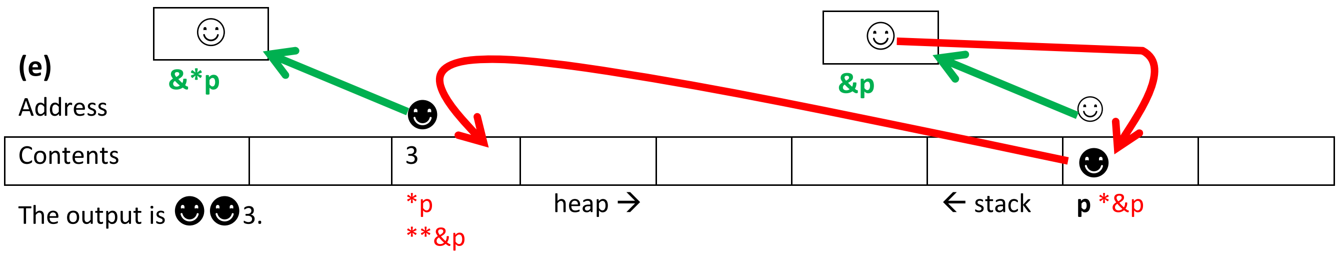
(d)



The output is 3.

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.

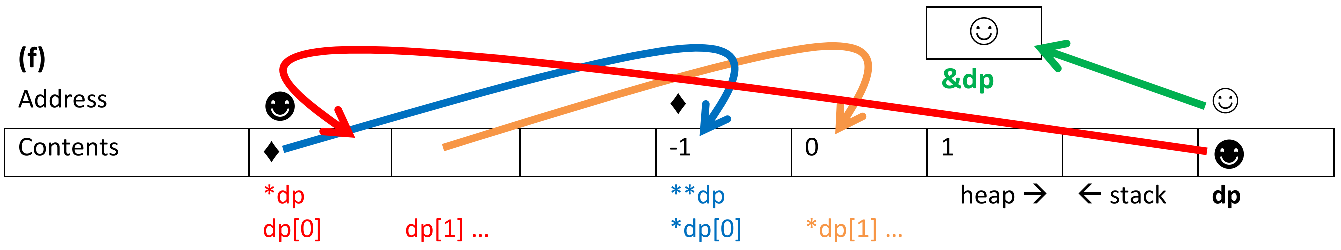
(e)



The output is ☻☻3.

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`).

(f)



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.

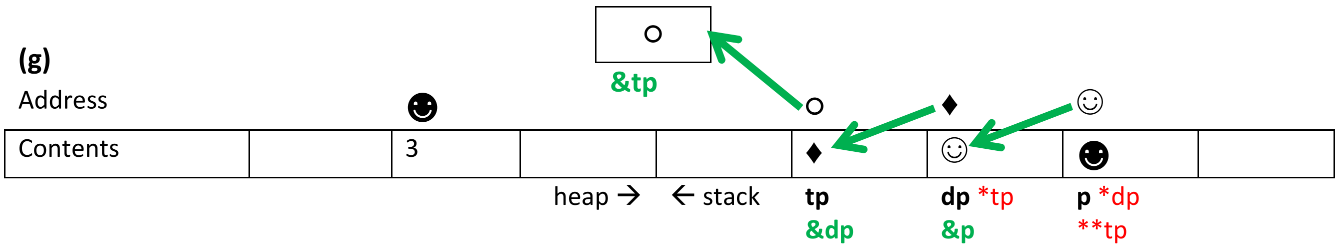
Just a note: `dp[0]` to `dp[2]` are in a contiguous block of memory because they are the elements of an array. However, `*dp[0]`, `*dp[1]` and `*dp[2]` are not guaranteed to be adjacent to each other.

If any of (b) - (g) is part of your program, after you are done with the integers and the array, do not forget to release them, as they were created on the heap using the `new` keyword.

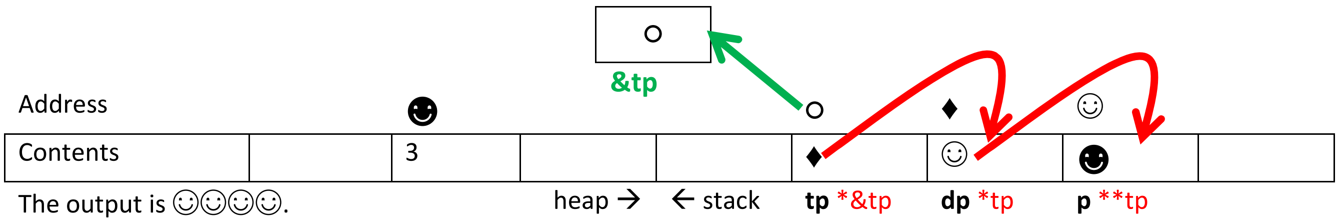
> Objects on the heap need to be released, as the system does not take back the memory they are located at, even when the function exits.

For example, in this part:
First, use a for loop to "delete" each pointer `for (int i = 0; i < 3; i++) delete dp[i];`
Remember, the memory at `dp[i]` does not get freed, but what is pointed to does, i.e. `*dp[i]`.
Then `delete []` dp to delete the array that dp points to.

(g)



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



The output is ☺☺☺☺.

Related Concepts (Q2)

- & results in a pointer
- * and [ ] dereferences
- Array pointers, double pointers
- Heap vs stack
- Freeing resources – "delete pointer"

3. *Object-Oriented Programming*

You want to print out the lyrics of this song[1], to teach (or confuse) kids about the sounds animals make:

```
Dog goes woof
Cat goes meow
Bird goes tweet
Mouse goes squeak
Cow goes moo
```

The lyrics can be generalized for different animals, each having a different *name* and **sound**. With knowledge of object-oriented programming, you want to demonstrate that it is possible to write a program that *displays* the song. To show that your program works, add the 5 animals above and test your program.

[1] Adapted from "The Fox" by Ylvis, 2013

Don't forget to include the necessary *system header*, and use the appropriate *namespace*. Use the skeleton on the next page to solve the problem.

```
class Animal {
     /* TODO: Implement data and functionality of an Animal here */
};
class Song {
  private:
     Animal** _animals;
     const int _size;
  public:
     Song() { /* TODO: Create your zoo, an Animal* array */ }
     ~Song() { /* TODO: Cleanup the 5 animals and the array... */ }
     void display() {
          for (int i = 0; i < _size; i++)
               cout << endl; /* TODO: Add the lyrics here... */
     }
};
int main() {
     Song song;
     song.display();
     return 0;
}
```

Related Concepts

- Class vs object
- Access modifiers
- Member variables & encapsulation
- Constructor & destructor
- Member functions, accessors, mutators
- Invoking member

Answer

Animal has 2 attributes: name and sound. These are modelled as string member variables – each Animal object has a name and sound. The data (member variables) is **encapsulated** by declaring them with the `private` access modifier. This means, outside the animal class, we *cannot directly access or modify* any Animal object's data.

`getName()` and `getSound()` are **member functions** serving as **accessors**. `getName()` returns the value of the `_name` member variable belonging to the Animal object pointed to by `this`. As we do not need to modify an animal's attributes, there are **no mutators** here.

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

class Animal {
  private:
     string _name; // e.g. Cow
     string _sound; // e.g. moo
  public:
     Animal(string name, string sound) {
          _name = name;
          _sound = sound;
     }
     string getName() { return _name; }
     string getSound() { return _sound; }
};
```

A constant `const` must be initialized, and it cannot be subsequently assigned a value. Therefore, we **use the initializer list** to provide a value to the constant `_size`.

```
class Animal {
     /* TODO: Implement data and functionality of an Animal here */
};

class Song {
  private:
     Animal** _animals;
     const int _size;
  public:
     Song() : _size(5) {
          _animals = new Animal* [_size];
          _animals[0] = new Animal("Dog", "woof");
          _animals[1] = new Animal("Cat", "meow");
          _animals[2] = new Animal("Bird", "tweet");
          _animals[3] = new Animal("Mouse", "squeak");
          _animals[4] = new Animal("Cow", "moo");
     }
     ~Song() {
          for (int i = 0; i < _size; i++)
               delete _animals[i];
          delete [] _animals;
     }
     void display() {
          for (int i = 0; i < _size; i++)
               cout << _animals[i]->getName() << " goes "
                    << _animals[i]->getSound() << endl;
     }
};
```

Remember to **return memory allocated on the heap**, and only memory allocated on the heap! Use the **`delete`** keyword **on the pointers** to those objects, and NOT on the objects themselves.

`_animals[i]->getName()` is equivalent to `(*_animals[i]).getName()`. First, `_animals[i]` moves from the (Animal pointer) array pointer to the (Animal pointer) array, `i` spaces from element 0. The result is an animal pointer. `*_animals[i]` moves from the Animal pointer to an Animal object. We then read the name of the Animal object using the `getName()` accessor member.

As we have mentioned in Q1 and Q2, **the first 100 times** you encounter some OOP code, **draw out what the code does in memory**…! Don't view code as mere text.

- Hope you had fun, prepare well for tutorial 2 ☻ -

Draw diagrams
Attempt tutorials
Test your solution