On C++ Pointers

Call Stack

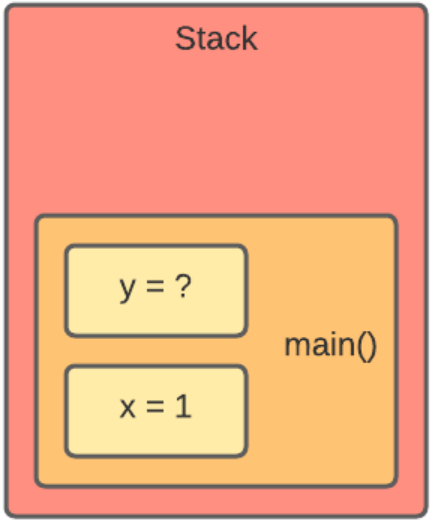
Modern OS typically divides the memory into several regions. The region that we will be focusing on for now is called the *call stack*.

When you run a program, every function invocation causes the OS to allocate some memory to store (among other things) the parameters passed into the function and the variables declared and used in the function. The memory allocated to each function call is called a *stack frame*. When a function returns, the stack frame is deallocated and freed up for other uses.

Let's look at the following example.

```
int main() {
    int x = 1;
    int y;
}
```

When the OS runs the previous program, it invokes, or calls the function `main`. A new stack frame is then created for `main()`. There are two variables `x` and `y` declared in main. Thus, the stack frame of the `main` will include these two variables. We initialize the `x` to 1 in the code above, so the value 1 will be placed in the memory location of `x`. The variable `y` remains uninitialized, so it will contain whatever value that happens to be in the memory location at the time.



Now, let's consider another program.

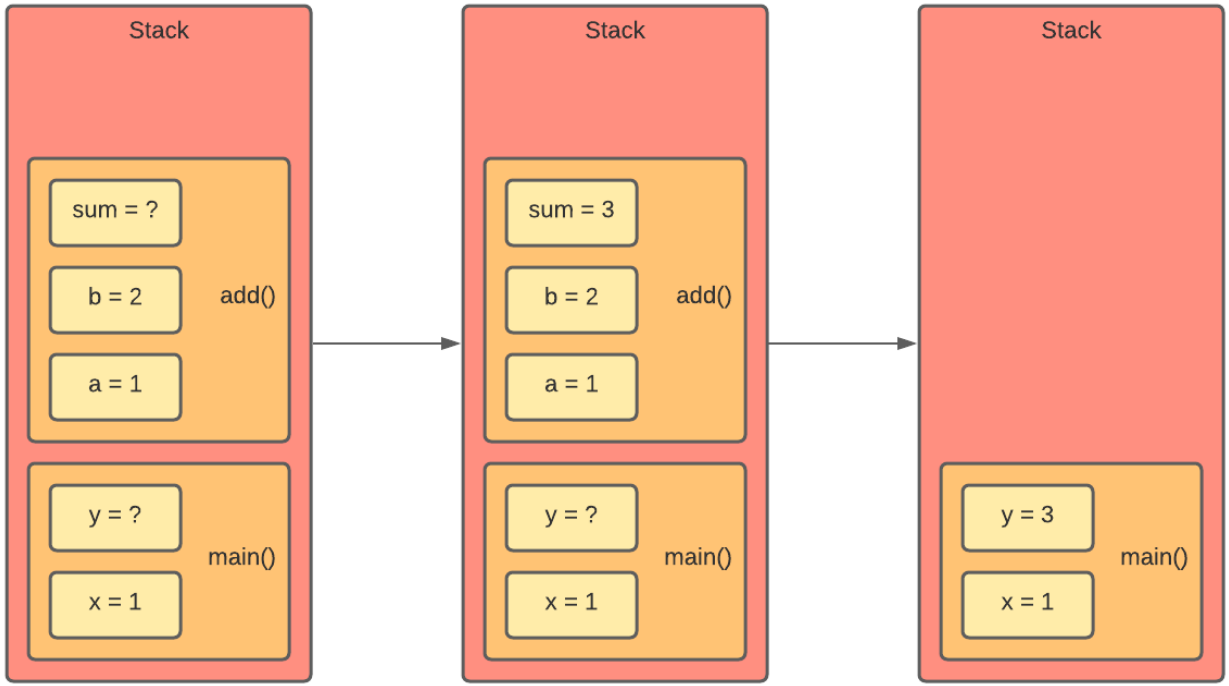
```
int add(int a, int b) {
    int sum;
    sum = a + b;
    return sum;
}

int main() {
    int x = 1;
    int y;
    y = add(x, 2);
}
```

Now, the program invokes the function add with two parameters, using `x` and `2` as arguments. The OS will then allocate another stack frame for `add`.

When the stack frame for `add` is created, `sum` is uninitialized, but `a` is initialized to whatever the value of `x` is when the function invoked (1 in this example), and `b` is initialized to `2`, since that is the argument passed into `add`.

After the stack frame for `add` is set up, the code is executed. The memory location for `sum` is then initialized to the sum of `a` and `b` (3 in this example), and the return statement is executed.



When the function returns, the stack frame for `add` is removed. The variables `sum`, `a`, `b` are deleted from the memory.

Notice how newer variables are always added on top of the older ones? This is an implementation of the *stack* data structure.

Heap

However, the problem with stack is that we cannot “free” a memory unless it is at the top of the stack. For example, you cannot allocate a dynamic array to the stack.

This is where the *heap** comes in. It is another important area of memory that the OS manages. Recall that a variable allocated on the stack has two properties:

1. Its lifetime is the same as the lifetime of the function the variable is declared in.
2. The memory allocation and deallocation are automatic.

The memory allocation on the heap can be done manually, but it is tricky. Before we exits the program, we must ensure that we have “free” all the memory that is allocated to the heap.

In C, the syntax to allocate and delete a memory to heap are `malloc`/`calloc` and `free`, respectively. C++, on the other hand, uses syntax such as `new` and `delete`.

* Not to be confused with the heap data structure, you will learn this in Week 8 of TIC2001.

Pointers

The address-of operator (&)

The address of a variable can be obtained by preceding its name with an ampersand sign (&), known as the *address-of operator*. For example,

```
foo = &var;
```

This would assign the address of variable `var` to `foo`; by preceding the name of the variable var with the *address-of operator* (&), we are no longer assigning the content of the variable itself to `foo`, but its address.

```
int x = 1;
cout << x << endl;      // prints 1
cout << &x << endl;     // prints the address of x
```

The dereference operator (*)

As just seen, a variable which stores the address of another variable is called a pointer. Pointers are said to “point to” the variable whose address they store. They can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (*).

Let’s look at the following example.

```
int a = 1;
int *b = &a;

cout << a << endl;          // prints 1
cout << &a << endl;         // prints 0x7fffeeaca40c
cout << *b << endl;         // prints 1
cout << b << endl;          // prints 0x7fffeeaca40c
cout << &b << endl;         // prints 0x7fffeeaca410
```

As you can see, `b` is simply a variable that stores the address of `a`. We can dereference `b` (by using `*b`) to access the value of `a`.

Also, notice that `&b != &a`. This is because **pointer is simply a variable that stores the address of another variable**, therefore it has its own address in the memory too.

Pointers arithmetic

To begin with, only addition and subtraction are allowed to be conducted on pointers. But both operations have a slightly different behavior (compared to regular variable), according to the size of the data type to which they point.

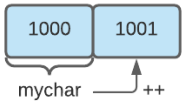
In general, `char` always has a size of 1 byte, `short` is usually larger than that, and `int` and `long` are even larger; the exact size of these being dependent on the system. For example, let's imagine that in a given system, `char` takes 1 byte, `short` takes 2 bytes, and `long` takes 4.

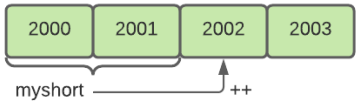
Suppose that we execute the following.

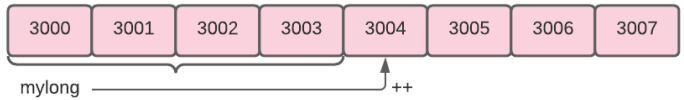
```
char *mychar;       // assume it points to memory location 1000
short *myshort;     // assume it points to memory location 2000
long *mylong;       // assume it points to memory location 3000

++mychar;
++myshort;
++mylong;
```

The pointers will now point to different memory locations.







Regarding the increment (++) and decrement (--) operators, they both can be used as either prefix or suffix of an expression, with a slight different in behavior: as a prefix, the increment happens before the expression is evaluated, and as a suffix, the increment happens after the expression is evaluated.

Consider the following scenarios.

```
*p++        // same as *(p++): increment pointer, and dereference unincremented address
*++p        // same as *(++p): increment pointer, and dereference incremented address
++*p        // same as ++(*p): dereference pointer, and increment the value it points to
(*p)++      // dereference pointer, and post-increment the value it points to
```

Confused? 😭

My advice:

Avoid using the increment (++) and decrement (--) operators on pointers. They make your code unnecessarily confusing and error-prone.

The dot (.) and arrow (->) operators

The dot (.) and arrow (->) operators are both used in C++ to access the members of a class, but they are used in different scenarios. Let's look at the following code.

```
class Point {
    public:
        int x;
        int y;

        Point(int x, int y) {
            this->x = x;
            this->y = y;
        }
};

int main() {
    Point a = Point(1, 2);
    cout << "Point a: (" << a.x << ", " << a.y << ")" << endl;      // prints "Point a: (1, 2)"
    Point *b = new Point(3, 4);
    cout << "Point b: (" << b->x << ", " << b->y << ")" << endl;    // prints "Point b: (3, 4)"
    delete b;       // remember to delete, otherwise it will cause memory leak
    return 0;
}
```

From the previous code, `a` is an object, therefore `a.x` and `a.y` are used to access the property of `x` and `y`, in which both are members of the object `a`.

On the other hand, `b->x` and `b->y` are essentially shorthand notations for `(*b).x` and `(*b).y`. In other words, `b` is a pointer to an object, therefore `b->x` and `b->y` are accessing the property `x` and `y` of the object that it points to.

But when should I use the *new* keyword when initializing an object?

Method 1 (`new`)

- Allocates memory for the object on the *heap*.
- Require you to `delete` your object later. If you don't delete it, then *memory leak* occurs.

Method 2 (not using `new`)

- Allocates memory for the object on the *stack*. Stack usually have less memory. If you allocate too many objects, *stack overflow* can occurs.
- You won't need to `delete` it later.

As for which method to use, you should choose the one that works the best on your scenario, given the constraints above.

References

- https://nus-cs1010.github.io/2021-s1/13-call-stack.html
- https://nus-cs1010.github.io/2021-s1/18-heap.html
- https://www.cplusplus.com/doc/tutorial/pointers/

Any feedbacks?

You can find me at e0550368@u.nus.edu