Introduction to Programming (CS 101)

Spring 2024

Lecture 15:

Recap of lecture 14 concepts + Introduction to pointers

Instructor: Preethi Jyothi

Based on material developed by Prof. Abhiram Ranade and Prof. Manoj Prabhakaran

Recap (I)

What is the output of the following program?

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

int main() {
  vector<int> nums = {1, 2};
  for(int i = 0; i < 4; i++)
    (i % 3 != 0) ? nums.push_back(i) : nums.pop_back();
  for(int x: nums)
    cout << x << " ";
}
```

1 2

1 1

2 1

2 2

Think of `push_back` and `pop_back` acting on `vector` as if it were a stack (i.e. last-in-first-out).

Recap (IIA)

What is the output of the following program?

```
int main() {
  struct entity {
    int x, y;
    int X(int z) {
      x += z; return x;
    }
  };
  entity e = {2, 3};
  e.x = e.X(e.y) + e.X(e.x);
  cout << e.x;
}
```

12

15

9

Compiler error

The expression `a + b` is evaluated left-to-right. Hence, the update to `e.x` via `e.X(e.y)` will reflect in the call `e.X(e.x)`. Thus, `e.x = 5 + 10 = 15`.

Recap (IIB) What is the output of the following program?

```
struct inner {
  int x, y;
};
struct outer {
  int x, y;
  inner i;
};
void setinner(outer& o) {
  (o.i).x += o.y;
  (o.i).y += o.x;
}
int main() {
  outer out = {4,3,{2,1}};
  setinner(out);
  cout << out.i.x << " " << out.i.y << endl;
}
```

4 3

5 5

2,1

Can access struct objects with nested '.' syntax. E.g., `(o.i).x`

`struct` example to check if a string is a palindrome

Demo; code struct-palindrome.cpp shared on Moodle

Recap (III)

Fill in the blank below, with a single statement, to get the following output:

```
1 2 3
3 4 5
7 8 9
```

```
void printvector(vector<vector<int>>& v) {
  for(vector<int> row: v) {
    for(int x: row)
      cout << x << " ";
    cout << endl;
  }
}

main_program {
  vector<vector<int>> A, B, C;
  A = {{1,2,3},{4,5,6},{7,8,9}};
  B = {{0,1,2},{3,4,5},{6,7,8}};

  C =
  printvector(C);
}
```

Can also write `{A.at(0),B.at(1),A.at(2)}`

Recall `A.at(i)` allows you to safely access vector elements (and throws an exception if you exceed the bounds).

Recap (IV) What does this program output?

```
string encodeString(const string& str) {
    string out = "";
    int count = 1;

    for (int i = 1; i <= str.size(); ++i) {
        if (i < str.size() && str[i] == str[i-1]) {
            count++;
        } else {
            out += str[i-1] + to_string(count);
            count = 1;
        }
    }
    return out;
}
main_program {
  string s;
  getline(cin, s); //aapppleeee
  cout << encodeString(s) << endl;
}
```

`to_string` is a function that takes an `int` or `float` as an argument and returns a string version of it.

output

```
a2p3l1e4
```

Introduction to Pointers

CS 101, 2025

Addresses

- Storage locations of variables have addresses
  - Exact address depends on the compiler and the operating system

```
int main(){
  int a;
  double b;
  int c;
  cout << &a << ", " << &b << ", "
       << &c << endl;
}
```

"Address of" operator
(not to be confused with reference type specifier)

Will print three distinct numbers (in hex).
The exact output is system dependent.
It can also vary across multiple runs (as a security measure!)

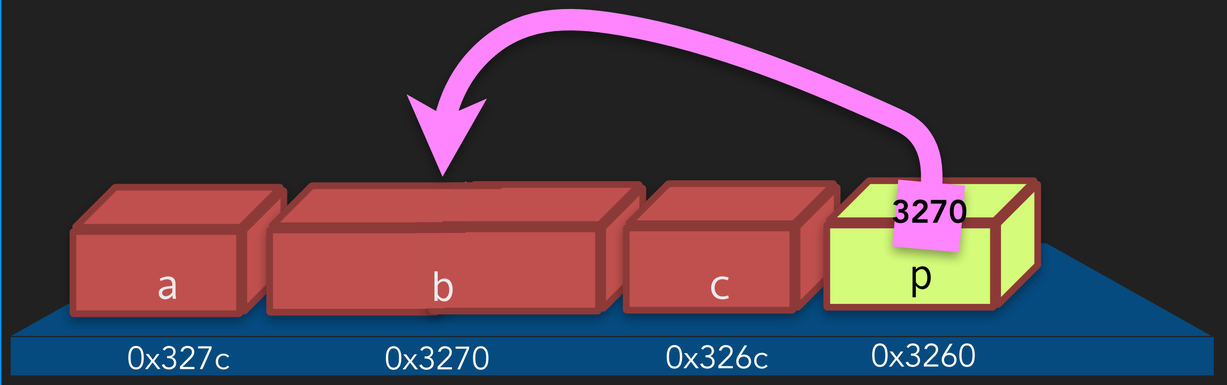
Pointers

- Pointer type variables can be used to store an address
- Declared as *type* name* (or *type * name* or *type *name*)
- Pointed location accessed as **name*

```
int main(){
  int a;
  double b;
  int c;
  double* p; // uninitialised!
  p = &b;    // assigned an address
  *p = 3.14; // now b==3.14
}
```

"Indirection" or "dereferencing" operator:
Follow the pointer
"Opposite" of the address-of operator

```
int a;
*(&a);  // same as a


int* p = &a;
&(*p);  // equals p
```



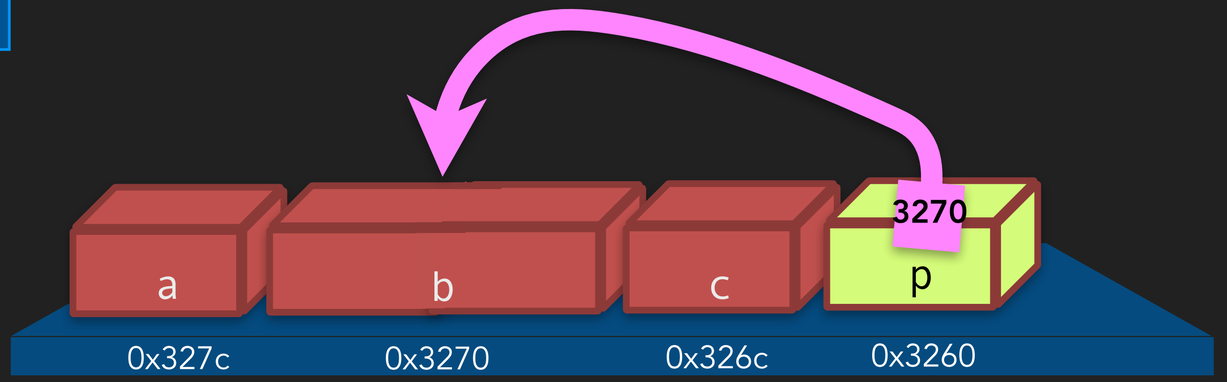
Pointers

- Pointer type variables can be used to store an address
- Declared as *type** *name* (or *type* * *name* or *type* **name*)
- Pointed location accessed as **name*

```
int a = 2, * p;
p = &a;
(*p)++; // now a==3
```

In a declaration statement with multiple variables, * is linked to the variable name, not the type name (similar to & in references)

Parentheses important here:
*p++ will be taken as *(p++)
which means something else
(coming up in the next class)



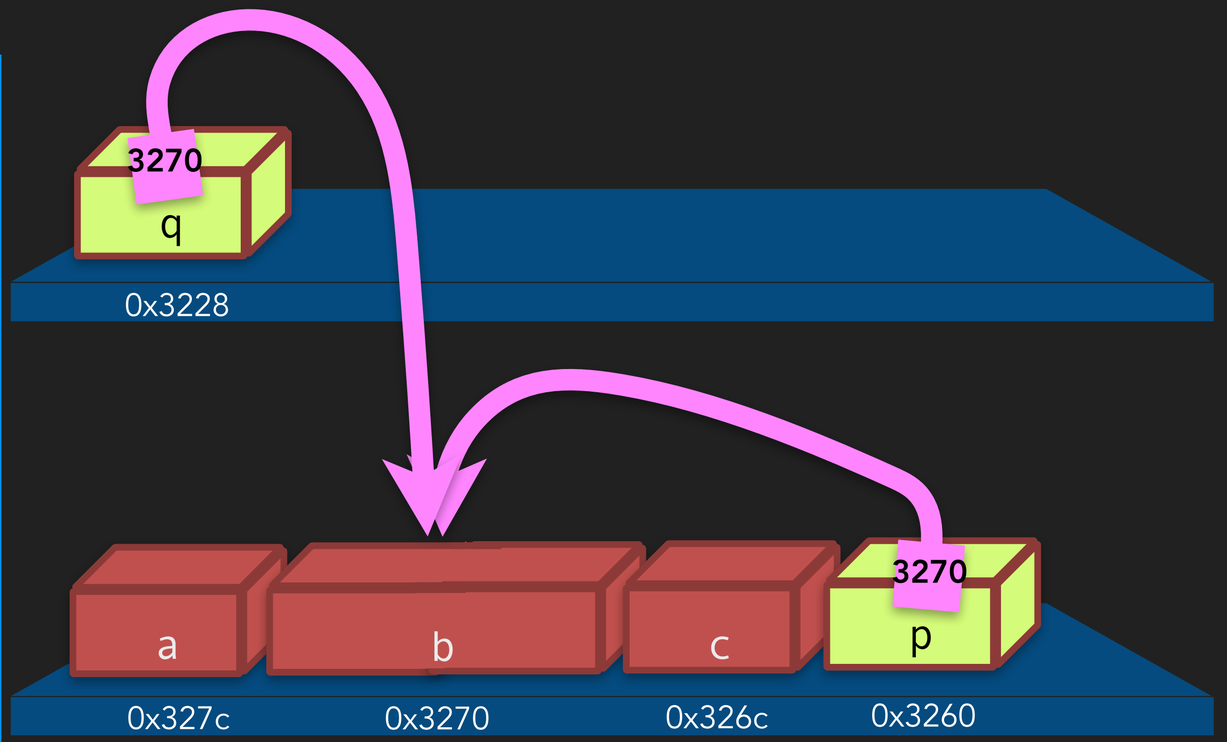
Example with pointers

```
main_program {
  int i = 1, j;
  bool b = true;
  int* p; p = &i;
  *p = 3; //*p as the LHS of an assignment means store a value into i
  j = *p; //*p as the RHS means use the value of i in place of *p
  cout << (*p) * j << endl; //prints 9 as output
  *p = b;
  cout << (*p) * i << endl; //prints 1 as output
}
```

Pointers

- Pointers can be passed as arguments to functions

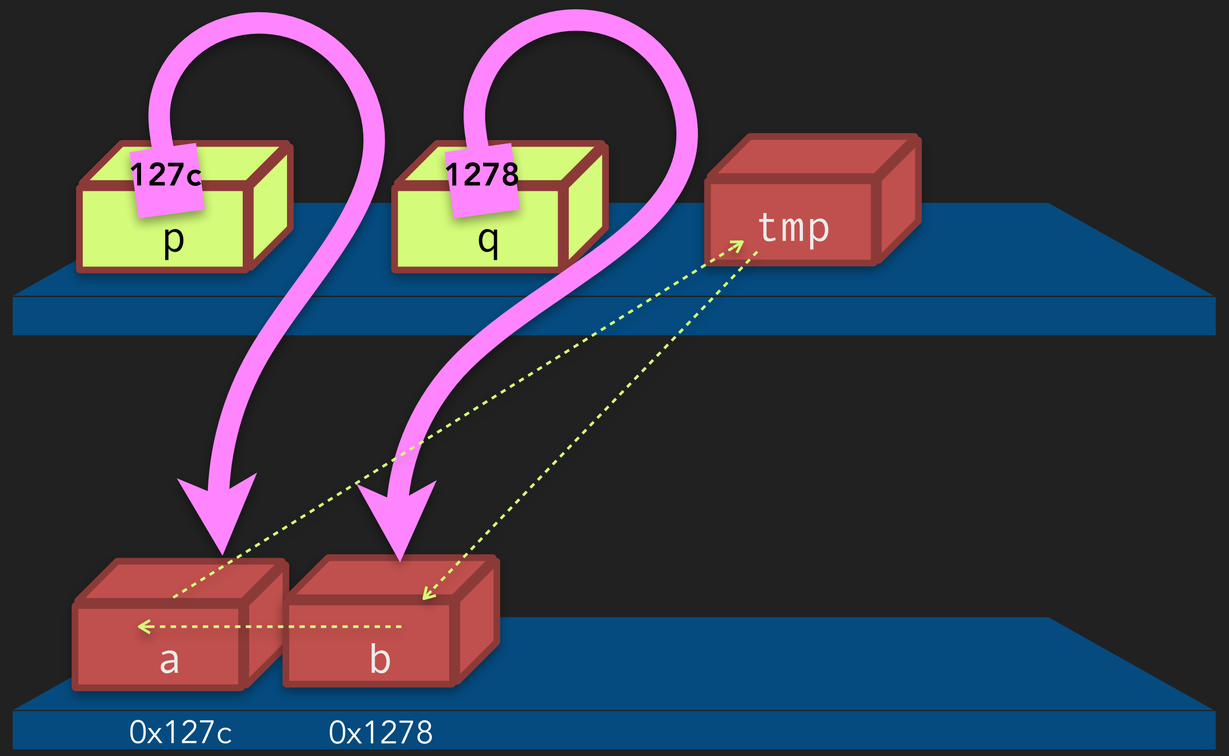
```
void f(double* q) { *q = 3.14; }

int main(){
  int a; double b, *p = &b; int c;
  f(&b);  // or, f(p)
  cout << b << endl;
}
```



Example: Swap Using Pointers

```
void swap(int* p, int* q){
  int tmp;
  tmp = *p;
  *p = *q;
  *q = tmp;
}
```

```
int main(){
  int a, b;
  ...
  swap(&a, &b);
  ...
}
```



Example

- Functions can return pointers too

```
int* max(int* p, int* q){
  return *p > *q ? p : q;
}
```

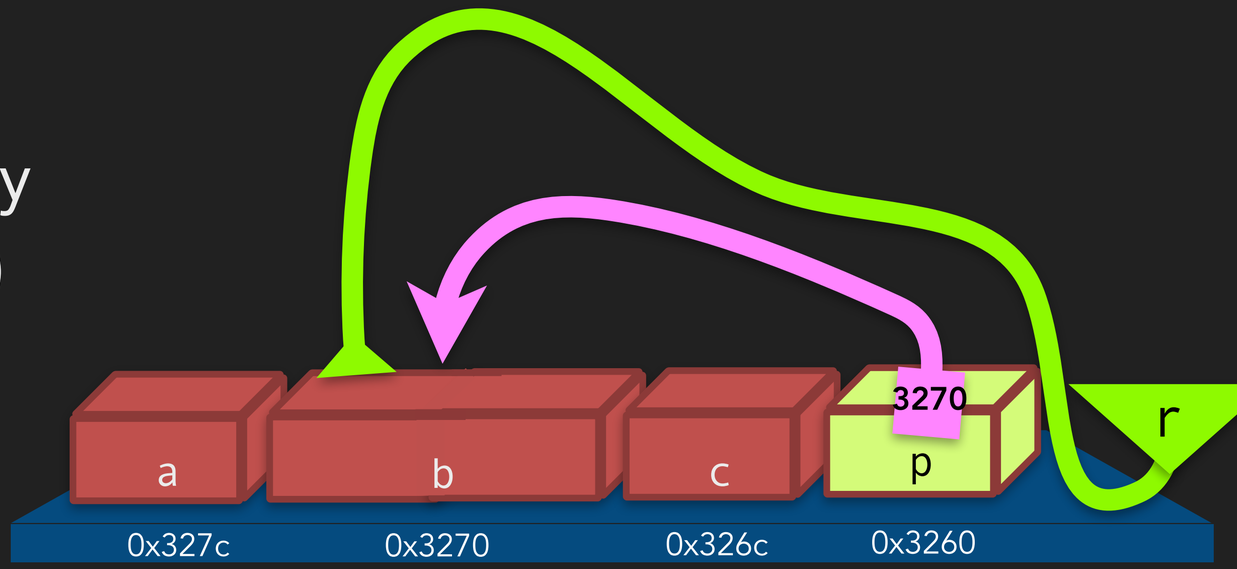
```
int main(){
  int a, b;
  ...
  *max(&a, &b) = 0; //set max to 0
  ...
}
```

Example with pointers in functions

```
int* f(int* x, int* y) {
  if(*x > *y) return x;
  else return y;
}

main_program {
  int p = 5, q = 4;
  *f(&p, &q) = 2;
  cout << p << " " << q << endl; // prints "2 4" as output
  cout << *f(&p, &q); // prints "4" as output
}
```

Pointers vs. References

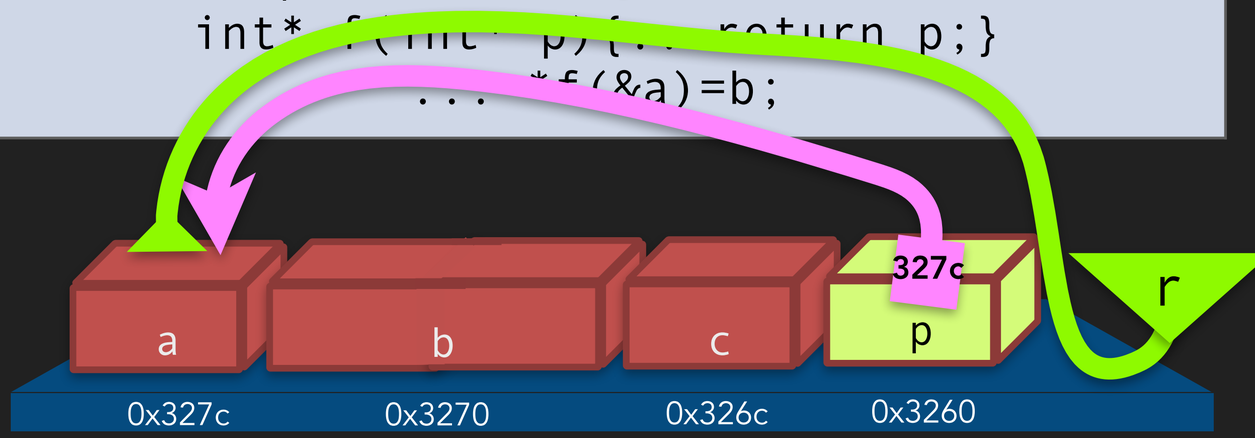
- References and pointers both allow accessing one variable via another
- Pointers are less strict about how they can be used, and can be manipulated more freely
  - Hence much more error prone!
    - Use references when possible
    - Or use objects from the standard library (internally implemented using pointers)
  - References are the safer (and more modern, compared to C) alternative to pointers in C++



Pointers vs. References

References	Pointers
Syntax: `int& r = a; r = 0;`	Syntax: `int* p = &a; *p = 0;`
Needs to be initialised: `int& r; // Error!`	Can be uninitialised: `int* p; // Allowed`
Cannot be "re-attached": `int& r=a; r=b;//value of b copied to a`	Can be "re-attached": `p=&a; ...; p=&b; // p now points to b`
Cannot be unattached	Can be set to `nullptr` to indicate that it is unattached

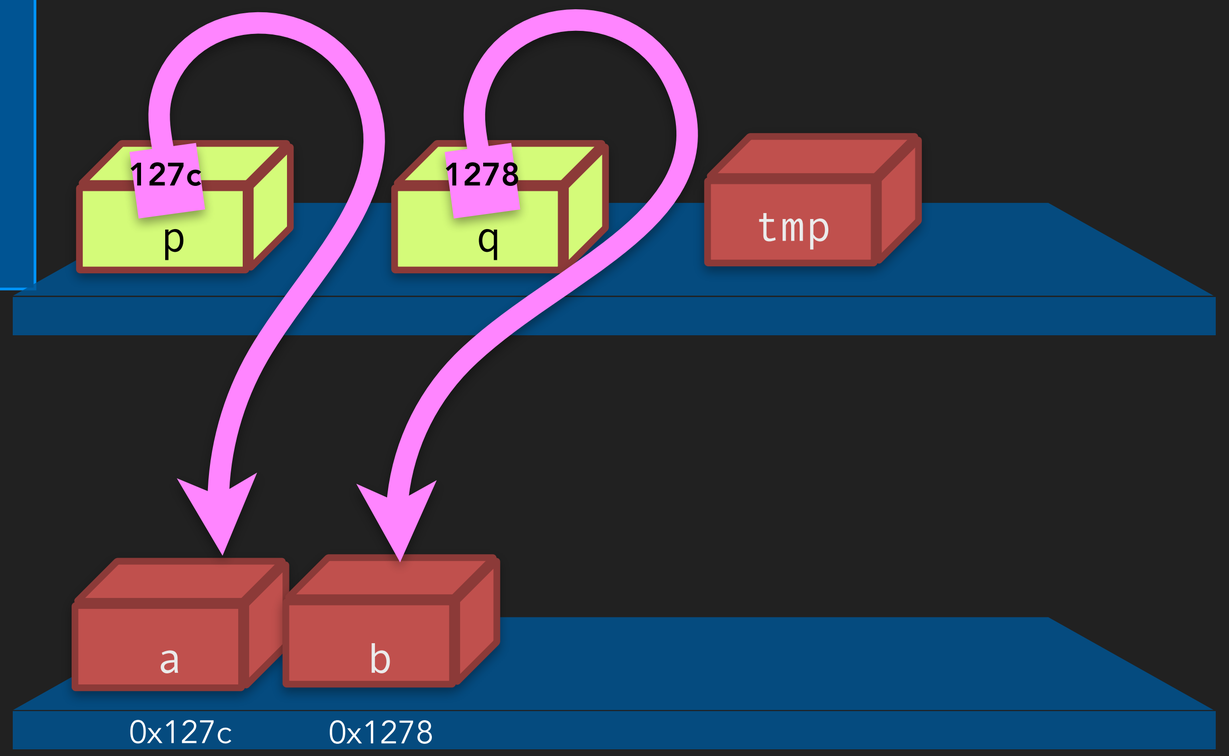
Can be passed as an argument and returned:
`int& f(int& r){.. return r;}`
`... f(a)=b;`
	Can be passed as an argument and returned:
	`int* f(int* p){.. return p;}`
	`... *f(&a)=b;`



Example: Swap Using Pointers

- Avoid swapping a variable with itself

```
void swap(bigStruct* p, bigStruct* q){
  if (p==q) return;
  bigStruct tmp = *p;
  *p = *q; *q = tmp;
}
```



- If references used, i.e.:
  `swap(bigStruct& a, bigStruct& b)`
  then also we can check `if(&a==&b)`
  - Address of a reference is the address of what it is referring to
  - Checking `if(a==b)` inspects the entire objects