An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 17

Pointers

Based on material developed by Prof. Abhiram G. Ranade

Recap

- Several programming concepts, so far

Data	Control Flow/Dynamics	Program Organization
Variables, expressions	Sequential execution	Statements, scope
Basic data types	(And sequence points)	`main()` and other functions
Internal representation	Conditional execution	Preprocessing
Reference variables	Conditional loops	Header files, Multiple C++ files
Structs	Function calls	Functions inside structs
Arrays	Lifetime of a variable	Function templates
From the Standard Library: I/O streams, `string`	Static variables	Namespaces
Pointers	Recursion	Classes (a glimpse)
More from the Standard Library	Exception handling	

- Additional important concepts coming up! ⬆️

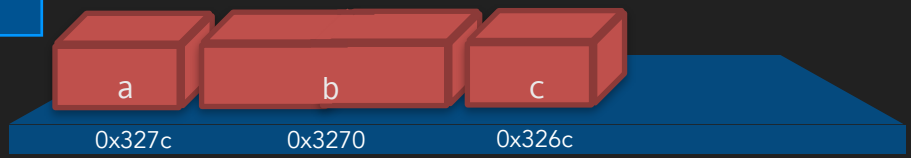
Addresses

- 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!)



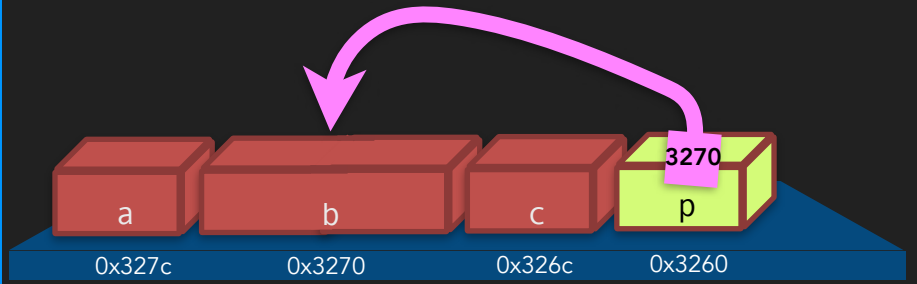
Pointers

- 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" operator:
Follow the pointer
"Opposite" of the address-of operator

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

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



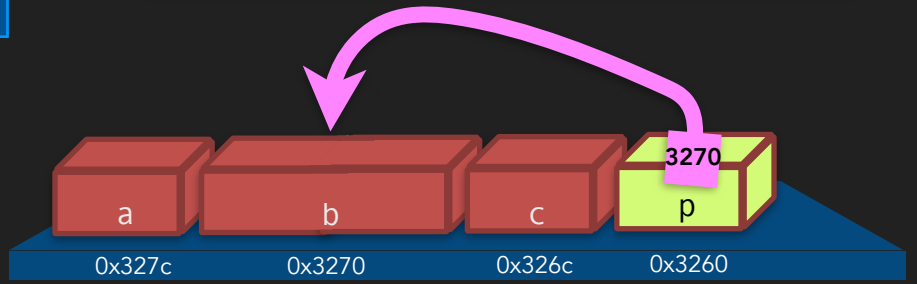
Pointers

- 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)

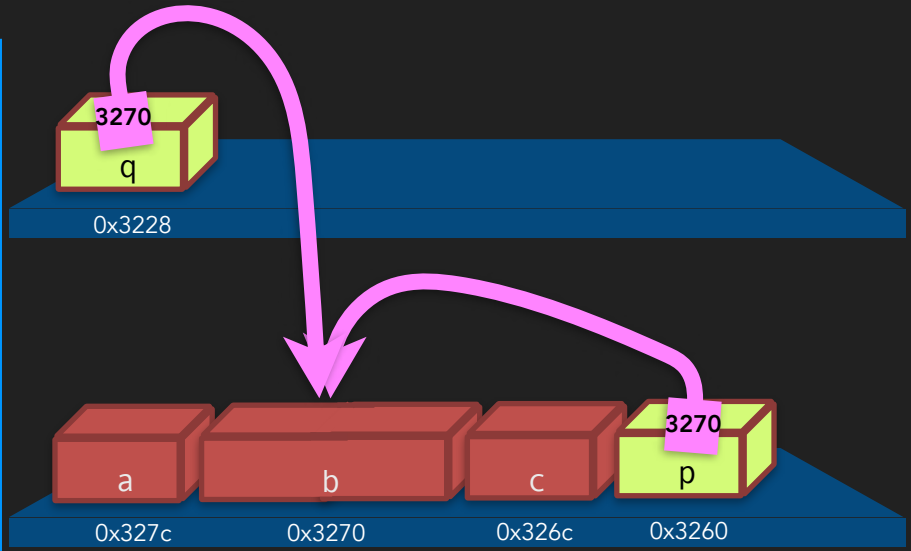
Parentheses important here:
*p++ will be taken as *(p++)
which means something else
(coming up soon)



Pointers

- Pointers can be passed as arguments to functions

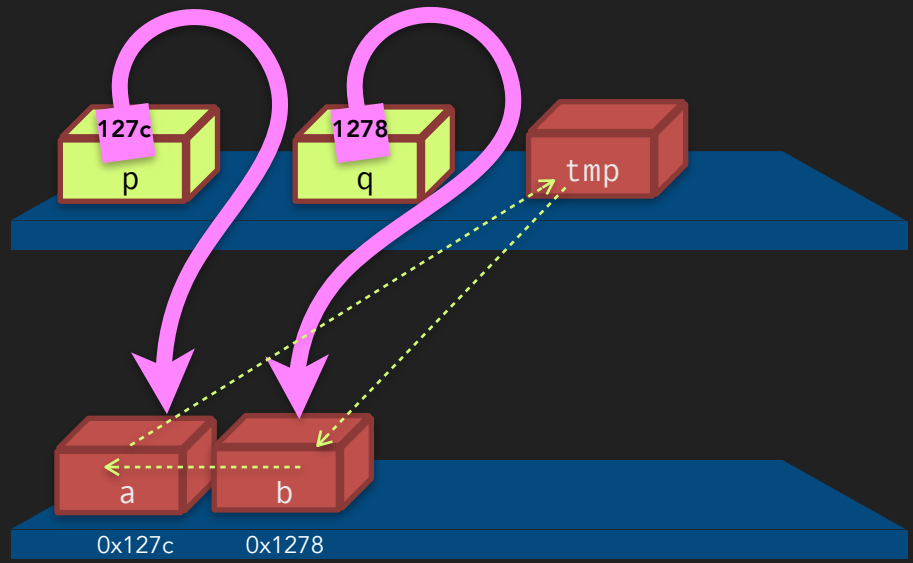
```
void f(double* q) { *q = 3.14; }

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

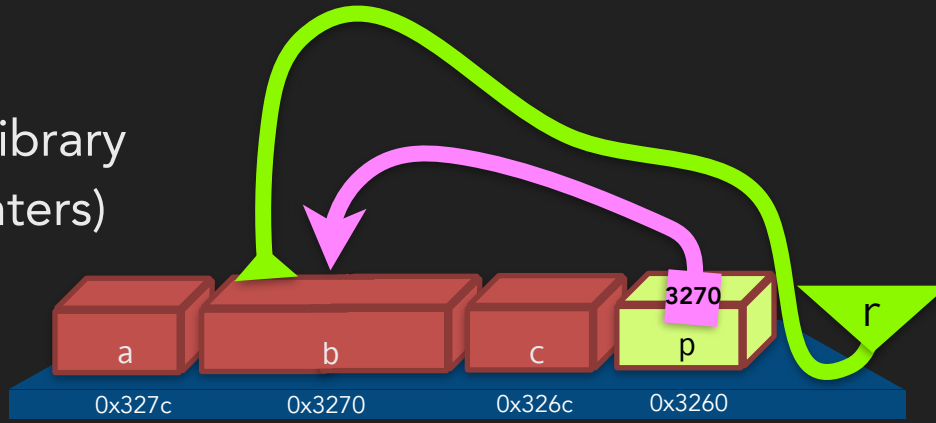
- Functions can return pointers too

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

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

Pointers vs. References

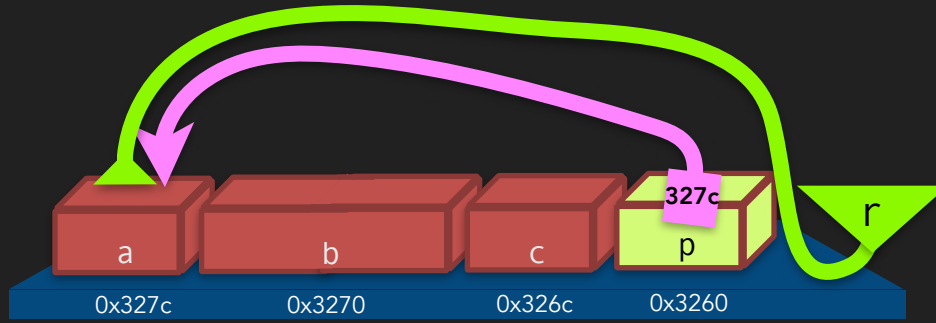
- References and pointers both allow accessing one variable via another
- Pointers are less strict about how they can be used, and can be manipulated more freely
  - Hence much more error prone!
    - Use references when possible
    - Or use objects from the standard library (internally implemented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

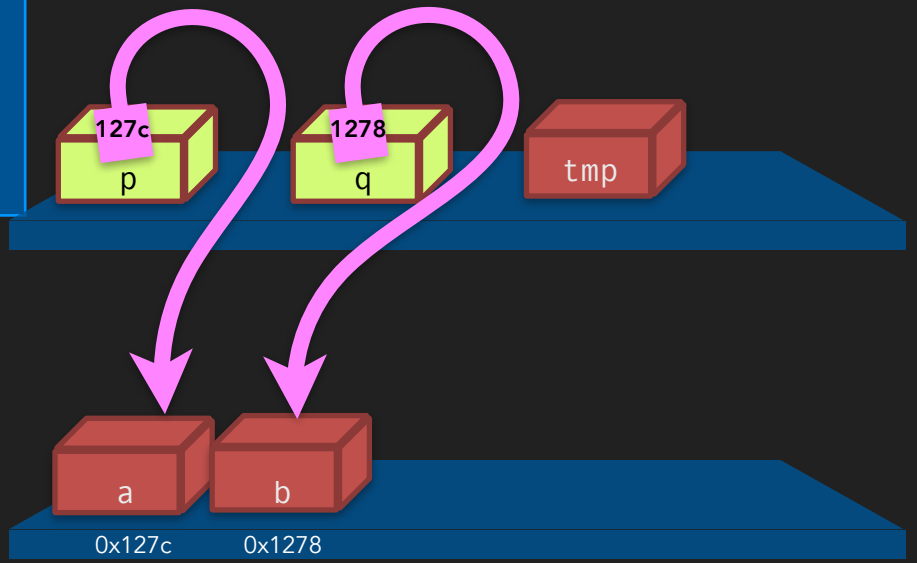
Can be passed as an argument and returned:
`int& f(int& r){.. return r;}`
`... f(a)=b;`
	Can be passed as an argument and returned:
	`int* f(int* p){.. return p;}`
	`... *f(&a)=b;`



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

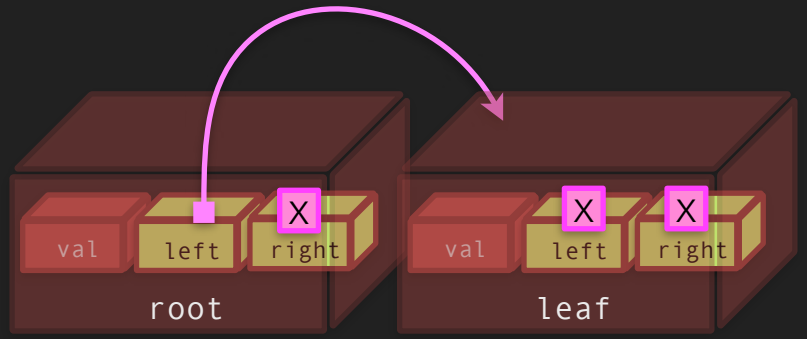
Example: Pointers in Structs

- Consider a struct to hold the information about a node in a binary tree

```
struct leafNode { int value; };
struct parentNode { int value; leafNode leftChild; leafNode rightChild};
struct grandParentNode { int value; parentNode leftChild; parentNode rightChild};
```

- Will need a different struct for nodes in each level of the tree!
- Problem: A node cannot contain itself
- Alternative, using pointers

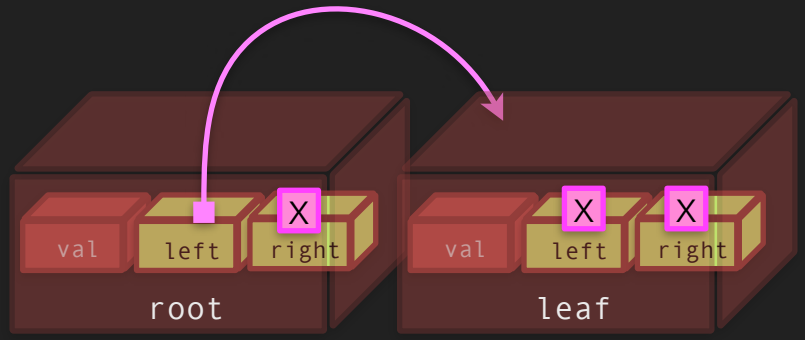
```
struct node {
  int value;
  node* leftChild;  // nullptr if no child
  node* rightChild; // nullptr if no child
};
...
node leaf = { 1, nullptr, nullptr };
node root = { 2, &leaf, nullptr };
...
```



Example: Pointers in Structs

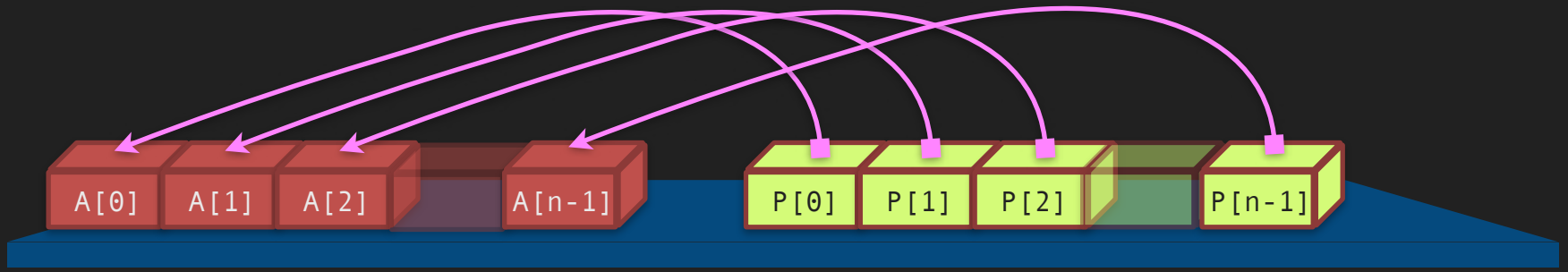
- Member X in a struct pointed to by p can be accessed as `p->X`
  - `p->X` is a shorthand for `(*p).X`

```
struct node {
  int val;
  node* left;  // nullptr if no child
  node* right; // nullptr if no child
};
void printTree(node* root) {
  if (!root)  // equivalently, if(root == nulptr)
    return;
  cout << " (" << root->val;
  printTree(root->left);
  printTree(root->right);
  cout << ") ";
}
```



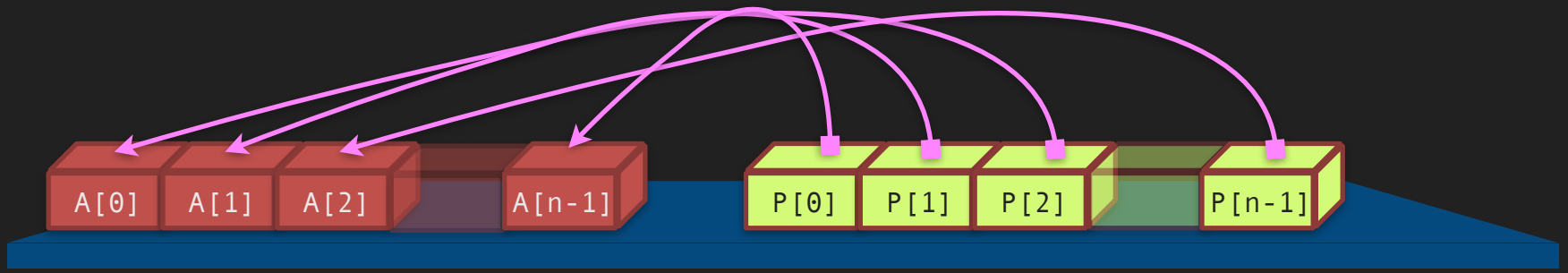
Example: Sorting Big Things

- Recall (merge-)sort: we copied elements around while merging arrays
- What if the elements are large?
- Idea: Don't move the actual elements. Sort pointers to the actual elements!



Example: Sorting Big Things

```
typedef std::string T;  // has a <= operator
void sort (T in[], T* out[], int left, int right, T* scratch[]) {
  if (left==right) { out[left] = &in[left]; return; } // base case
  int mid = (left+right)/2;
  sort(in,scratch,left,mid,out);
  sort(in,scratch,mid+1,right,out);
  merge(scratch,out,left,right);
}
```



Example: Sorting Big Things


```
typedef std::string T;  // has a <= operator
```

```
void sort (T in[], T* out[], int left, int right, T* scratch[]) {
  if (left==right) { out[left] = &in[left]; return; } // base case
  int mid = (left+right)/2;
  sort(in,scratch,left,mid,out);
  sort(in,scratch,mid+1,right,out);
  merge(scratch,out,left,right);
}
```

```
// merge X[left..mid] and X[mid+1..right] into Y[left..right], where mid = (left+right)/2
void merge(T* X[], T* Y[], int left, int right) {
    int mid = (left+right)/2, L = left, R = mid+1;   // L,R: next indices of left/right halves
    for(int i=left; i <= right; ++i) {
        if(L <= mid && (R > right || *X[L] <= *X[R])) Y[i] = X[L++];  // copy from left
        else Y[i] = X[R++];                                           // copy from right
    }
}
```

Arrays as Pointers

- Recall that C-style arrays can be passed to functions (syntactically by value, but semantically by reference)
  - Then the array is implicitly converted to a pointer to the first element of the array

```
void f(int A[]);
```

same as

```
void f(int* A);
```

- Given any pointer, can use array-like indexing
  - Works correctly if the pointer is indeed pointing to the first element of an array

```
int* p;
p = A;
p[2] = 1;
```

Pointer Arithmetic

- Adding/subtracting integers, incrementing/decrementing, and comparisons are allowed on pointers, by interpreting them as pointing to elements in an array
  - `p+i` is the same as `& p[i]`
    - And, `*(p+i)` same as `p[i]`
  - `p<q` if `p == A+i` and `q == A+j` where `i<j`

```
void printArray(int* p, int* q) { while(p<q) cout << *p++ << " "; }
```

```
int A[10];
...
printArray(A,A+10);
```

Pointers to Pointers

- A pointer type variable is just like any other variable that has an address associated with it
  - cf. References don't have their own addresses: `int& r = a; &r == &a;`
- It can have another pointer pointing to it

```
int* p;
int** pp = &p;
```

- An array of pointers as a function parameter can be equivalently written as a pointer to a pointer

```
void sort (T in[], T* out[], int left, int right, T* scratch[]);
void merge(T* X[], T* Y[], int left, int right);
```


```
void sort (T* in, T** out, int left, int right, T** scratch);
void merge(T** X, T** Y, int left, int right);
```

Summary

- Address of variables, Pointer type variables, Indirection operator
- Pointers vs. references: Similarities and differences (use references when you can)
- Pointers in structs
- Arrays as pointers
- Pointer arithmetic
- Pointers to pointers
- Next: Venturing outside the stack, guided by pointers

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

```
struct node { ... node* left;..};
... node* root; ... root -> left;
```