Why should I use pointers when I can access objects directly?

Why should I use pointers when I can access objects directly?

- We have seen applications of pointers in memory management

Why should I use pointers when I can access objects directly?

The following example will explain this part.

```
typedef struct pair{
        int x[512];
        int y[512];
} pair;
```

Consider a large compound data type ‘pair’

```
. . .
pair add1(pair a, pair b){
        pair temp;
        int i;
        for(i=0; i<512; i++){
                temp.x[i] = a.x[i]+b.x[i];
                temp.y[i] = a.y[i]+b.y[i];
        }
        return temp;
}
int main(){
        int i;
        pair a, b, c;
        …
        //approach1
        c = add1(a, b);
        …
}
```

Approach 1:
Compute c by passing objects

```
. . .
void add2(pair *p0, pair *p1, pair *p2){
          int i;
            for(i=0; i<512; i++){
              p2->x[i] = p0->x[i] + p1->x[i];
              p2->y[i] = p0->y[i] + p1->y[i];
            }
          return;
}
int main(){
          int i;
          pair a, b, c;
          pair *p0, *p1, *p2;
          p0 = &a; p1=&b; p2=&c;
          …
          add2(p0, p1, p2);
          …
}
```

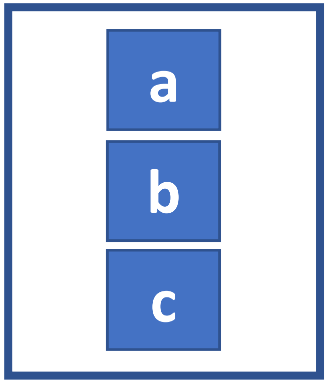
Approach 2:
Compute c by passing pointers

Both approaches compute the same result.

Question:
Which one would be better for a system?

```
. . .
pair add1(pair a, pair b){
        pair temp;
        int i;
        for(i=0; i<512; i++){
                temp.x[i] = a.x[i]+b.x[i];
                temp.y[i] = a.y[i]+b.y[i];
        }
        return temp;
}
int main(){
        int i;
        pair a, b, c;
        ...
        //approach1
        c = add1(a, b);
        ...
}
```

Approach 1:
Compute c by passing objects

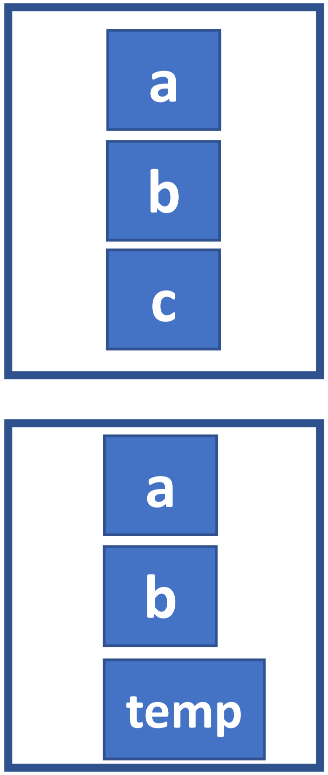


Stack frame of main()

Initially large objects a, b, c are in stack frame of main()

```
. . .
pair add1(pair a, pair b){
        pair temp;
        int i;
        for(i=0; i<512; i++){
                temp.x[i] = a.x[i]+b.x[i];
                temp.y[i] = a.y[i]+b.y[i];
        }
        return temp;
}
int main(){
        int i;
        pair a, b, c;
        ...
        //approach1
        c = add1(a, b);
        ...
}
```

Approach 1:
Compute c by passing objects



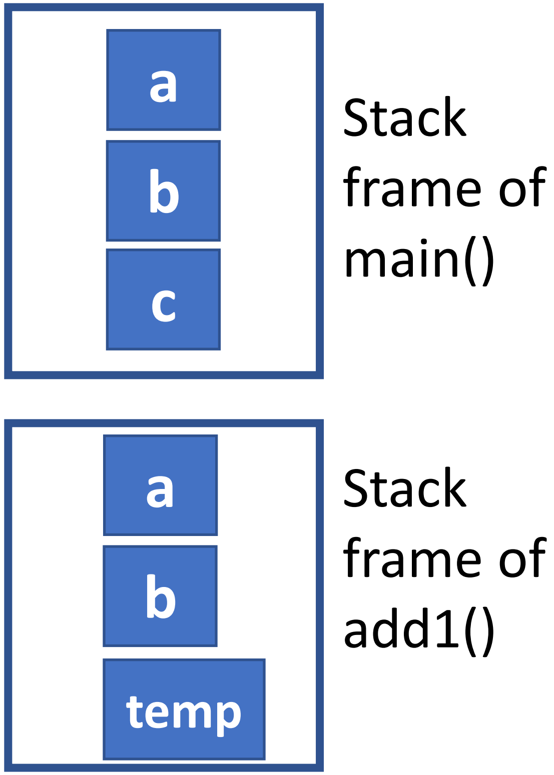
Stack frame of main()

Stack frame of add1()

add1() is called and then **large** a and b are passed.
→ they are **copied**.

```
. . .
pair add1(pair a, pair b){
        pair temp;
        int i;
        for(i=0; i<512; i++){
                temp.x[i] = a.x[i]+b.x[i];
                temp.y[i] = a.y[i]+b.y[i];
        }
        return temp;
}
int main(){
        int i;
        pair a, b, c;
        ...
        //approach1
        c = add1(a, b);
        ...
}
```

Approach 1:
Compute c by passing objects

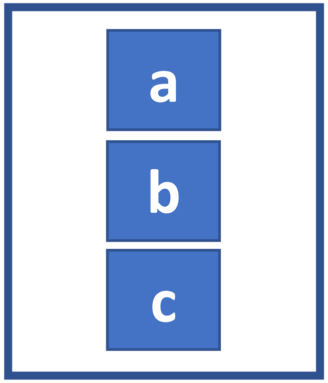


In the end add1() returns **large** ‘temp’.
→ It is copied into c

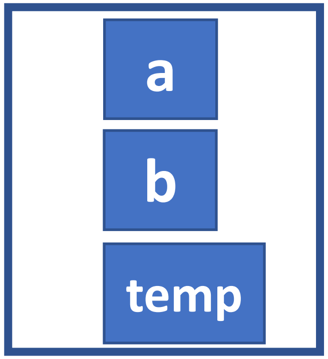
Approach 1:
Compute c by passing objects

```
. . .
pair add1(pair a, pair b){
        pair temp;
        int i;
        for(i=0; i<512; i++){
                temp.x[i] = a.x[i]+b.x[i];
                temp.y[i] = a.y[i]+b.y[i];
        }
        return temp;
}
int main(){
        int i;
        pair a, b, c;
        …
        //approach1
        c = add1(a, b);
        …
}
```

Lots of big-data copy happen.



Stack frame of main()

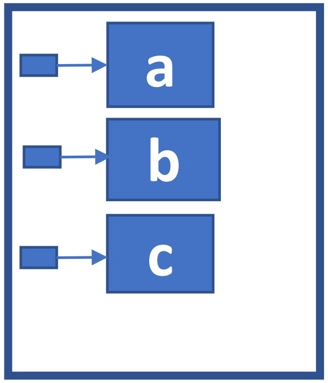


Stack frame of add1()

In the end add1() returns **large** ‘temp’.
→ It is copied into c

```
. . .
void add2(pair *p0, pair *p1, pair *p2){
        int i;
          for(i=0; i<512; i++){
            p2->x[i] = p0->x[i] + p1->x[i];
            p2->y[i] = p0->y[i] + p1->y[i];
          }
        return;
}
int main(){
        int i;
        pair a, b, c;
        pair *p0, *p1, *p2;
        p0 = &a; p1=&b; p2=&c;
        ...
        add2(p0, p1, p2);
        ...
}
```

Approach 2:
Compute c by passing pointers

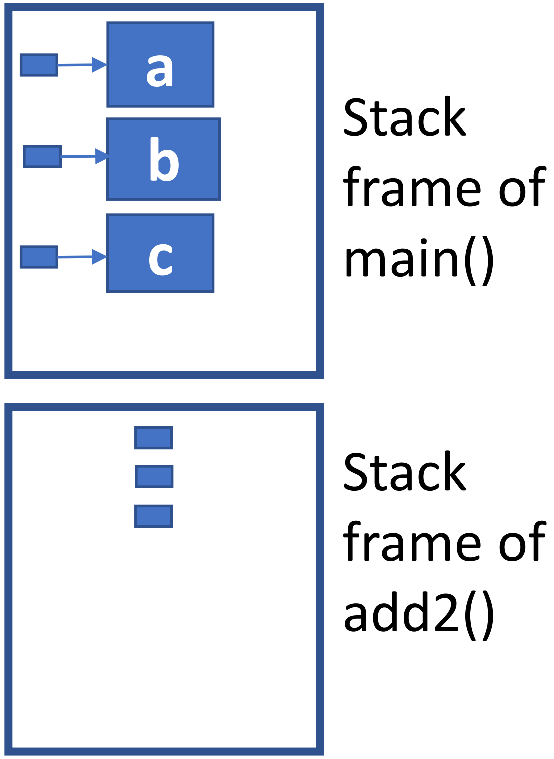


Stack frame of main()

Initially large objects a, b, c and 8-byte pointers are in stack frame of main()

```
. . .
void add2(pair *p0, pair *p1, pair *p2){
          int i;
            for(i=0; i<512; i++){
              p2->x[i] = p0->x[i] + p1->x[i];
              p2->y[i] = p0->y[i] + p1->y[i];
            }
          return;
}
int main(){
          int i;
          pair a, b, c;
          pair *p0, *p1, *p2;
          p0 = &a; p1=&b; p2=&c;
          …
          add2(p0, p1, p2);
          …
}
```

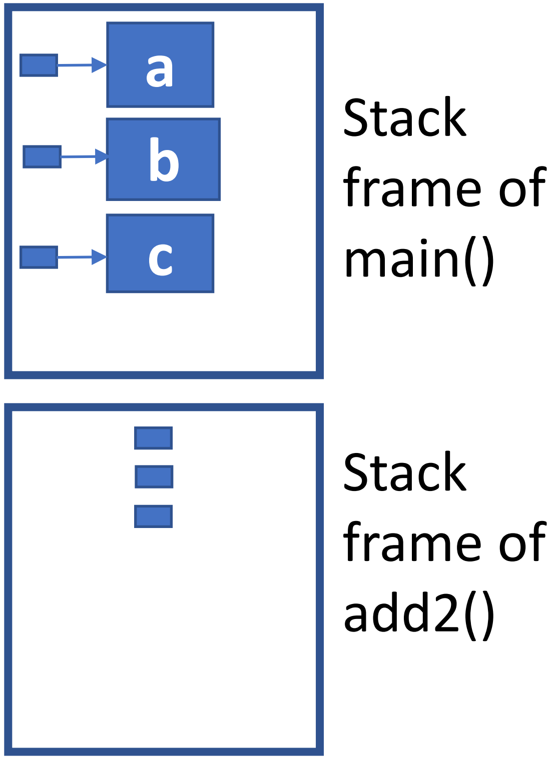
Approach 2:
Compute c by passing pointers



add2() is called and then pointers are passed.
→ **Small** 8-byte pointers are **copied**

```
. . .
void add2(pair *p0, pair *p1, pair *p2){
          int i;
            for(i=0; i<512; i++){
              p2->x[i] = p0->x[i] + p1->x[i];
              p2->y[i] = p0->y[i] + p1->y[i];
            }
          return;
}
int main(){
          int i;
          pair a, b, c;
          pair *p0, *p1, *p2;
          p0 = &a; p1=&b; p2=&c;
          ...
          add2(p0, p1, p2);
          ...
}
```

Approach 2:
Compute c by passing pointers



add2() updates c directly

So, overall only 3 pointers are copied!

Conclusions: pass-by-value vs pass-by-pointer

- Pass-by-value copies objects from one stack frame to other
- Pass-by-pointer copies only pointers

Thus, pass-by-pointer is more efficient for large data objects