MODULE 1 / UNIT 6

PASSING BY VALUE, REFERENCE, OR POINTER

Today

- Argument passing in python, R, C++, and R/C++
- Scope of variable
- Shallow copy and deep copy
- Passing arguments by value vs. reference
- Passing arguments by pointer

A simple python code

```
def valPy(x):
    x += 1
```

```
def vecPy(x):
    x[0] += 1
```

```
y = 1
valPy(y)
print (y)
```

```
z = [1, 2]
vecPy(z)
print (z)
```

?

?

A simple python code

```
def valPy(x):
    x += 1
```

```
def vecPy(x):
    x[0] += 1
```

```
y = 1
valPy(y)
print (y)
```

```
z = [1, 2]
vecPy(z)
print (z)
```

```
1
```

```
[2, 2]
```

why?

A similar R code

```
valR <- function(x) {
  x <- x+1
}
```

```
y <- 1
valR(y)
print(y)
```

?

```
vecR <- function(x) {
  x[1] <- x[1] + 1
}
```

```
z <- c(1,2)
vecR(z)
print(z)
```

?

A similar R code

```
valR <- function(x) {
  x <- x+1
}
```

```
y <- 1
valR(y)
print(y)
```

```
[1] 1
```

```
vecR <- function(x) {
  x[1] <- x[1] + 1
}
```

```
z <- c(1,2)
vecR(z)
print(z)
```

```
[1] 1 2
```

why?

Argument passing in typical C++

```
#include <Rcpp.h>
#include <vector>
#include <iostream>
using namespace Rcpp;
using namespace std;

void valC(double x) {
  ++x;
}

void vecC(vector<double> x) {
  ++x[0];
}
```

```
// [[Rcpp::export]]
void arg_test() {
  double y = 1;
  vector<double> z(2);
  z[0] = 1; z[1] = 2;

  valC(y);
  cout << y << endl;

  vecC(z);
  cout << z[0] << " " << z[1] << endl;
}
```

Results of C++ example

```
arg_test();
```

```
1
1 2
```

why?

How about between R and C++?

```
#include <Rcpp.h>
#include <vector>
using namespace Rcpp;
using namespace std;
// [[Rcpp::export]]
void valRcpp(double x) {
  ++x;
}

// [[Rcpp::export]]
void vecRcpp(NumericVector x) {
  ++x[0];
}
```

```
y <- 1
valRcpp(y)
print(y)
```

?

```
z <- c(1,2)
vecRcpp(z)
print(z)
```

?

How about between R and C++?

```
#include <Rcpp.h>
#include <vector>
using namespace Rcpp;
using namespace std;
// [[Rcpp::export]]
void valRcpp(double x) {
  ++x;
}

// [[Rcpp::export]]
void vecRcpp(NumericVector x) {
  ++x[0];
}
```

```
y <- 1
valRcpp(y)
print(y)
```

```
[1] 1
```

```
z <- c(1,2)
vecRcpp(z)
print(z)
```

```
[1] 2 2
```

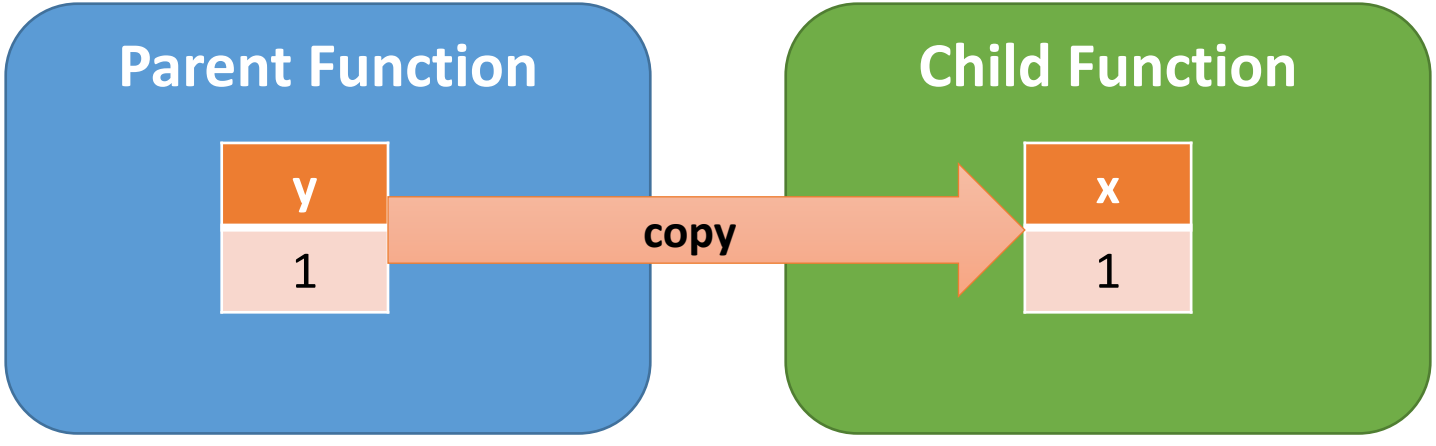
why?

Observations so far..

- **When passing a single numeric value as an argument..**
  - All examples ignored the updated variables
- **When passing a vector as an argument..**
  - R-only and C++ only examples ignored the updates
  - python and R/C++ examples changed the original vector.

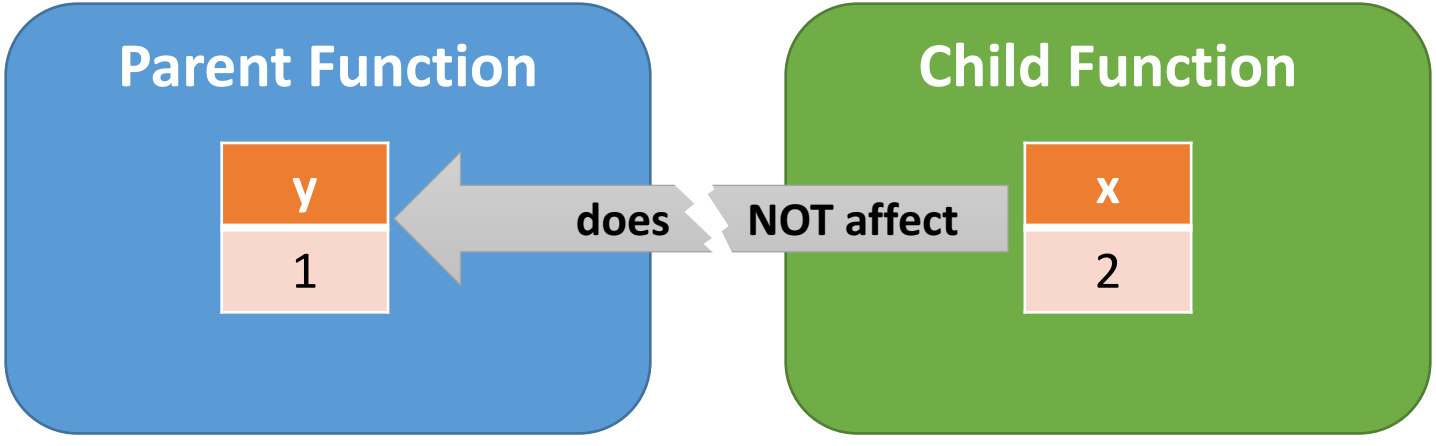
Basics of argument passing

- Across python, R, and C++, when function calls are made, arguments are being copied by default



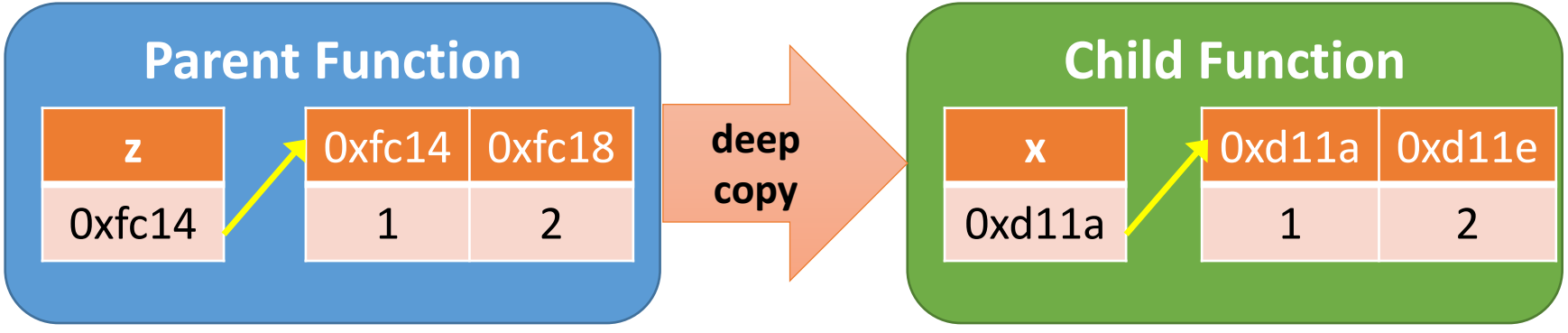
Basics of argument passing

- When copied arguments are updated within the child function, the update does not affect the original value within the scope the parent function.

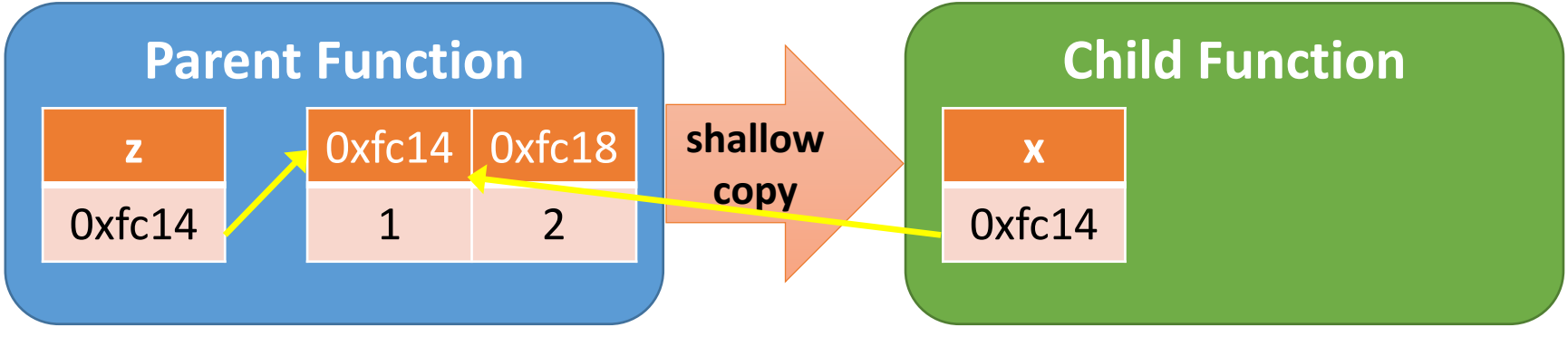


Shallow vs. deep copy of complex objects

- Deep copy make a complete “clone” of a complex object

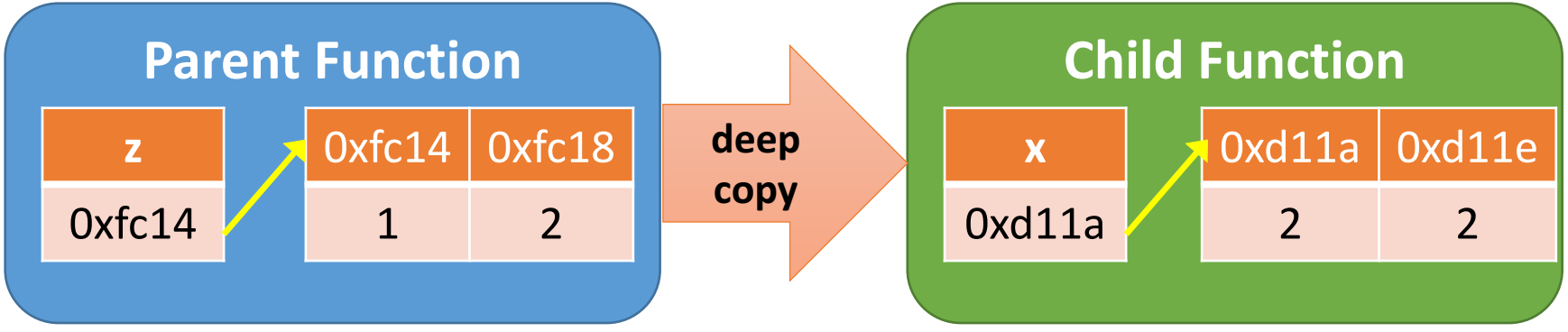


- Shallow copy only make a copy of ‘first-level’ data

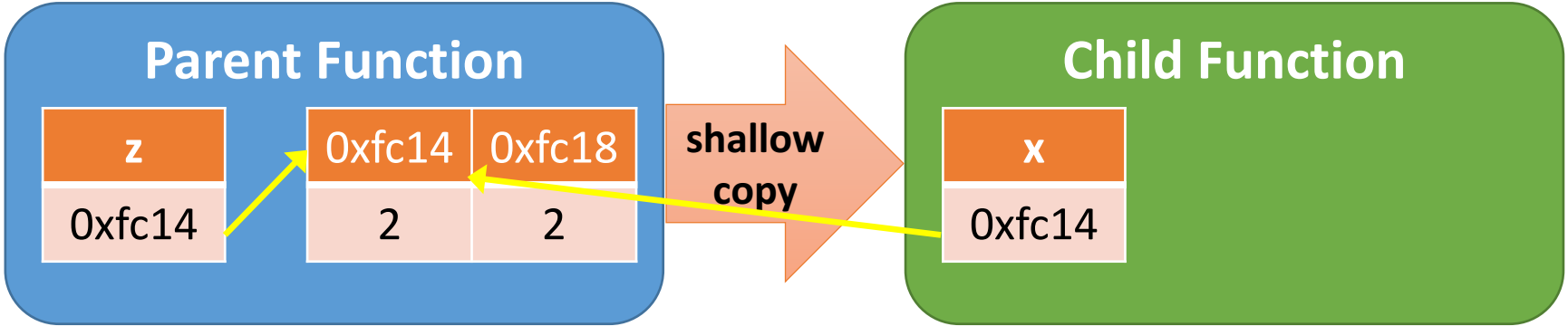


Updates with shallow vs. deep copy

- With deep, updates only affects the contents of the copy



- Updates on shallow-copied object affects the original copy



Which one is better?

- **Deep copy is..**
  - arguably less confusing (i.e. no unexpected updates)
  - heavier due to cloning of (potentially) a large object
  - not too slow if "smart" cloning (only when object is updated) is used.
- **Shallow copy is..**
  - arguably less confusing, due to a simple rule
  - lighter by avoiding cloning overhead
  - potentially cumbersome because explicit cloning is required when needed.

Shallow vs deep copies in different languages

- **Shallow copy is used in..**
  - Python (list, dictionary, tuple, …)
  - R/C++ (NumericVector, NumericMatrix, …)

- **Deep copy is used in…**
  - R
  - STL class in C++
    - For other types, it may vary
    - There are multiple modes, such as “call-by-reference”

A slight modification of the C++ example

```
#include <Rcpp.h>
#include <vector>
#include <iostream>
using namespace Rcpp;
using namespace std;

void valCRef(double& x) {
  ++x;
}

void vecCRef(vector<double>& x) {
  ++x[0];
}
```

```
// [[Rcpp::export]]
void arg_testRef() {
  double y = 1;
  vector<double> z(2);
  z[0] = 1; z[1] = 2;

  valCRef(y);
  cout << y << endl;

  vecCRef(z);
  cout << z[0] << " " << z[1] << endl;
}
```

The results are totally different..

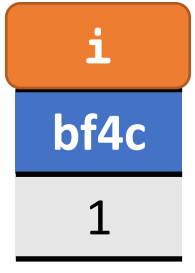
```
arg_testRef();
```

```
2
2 2
```

why?

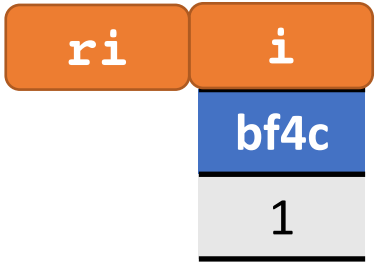
Pointers and references in C++

```
int  i  = 1;  // i is an integer
```



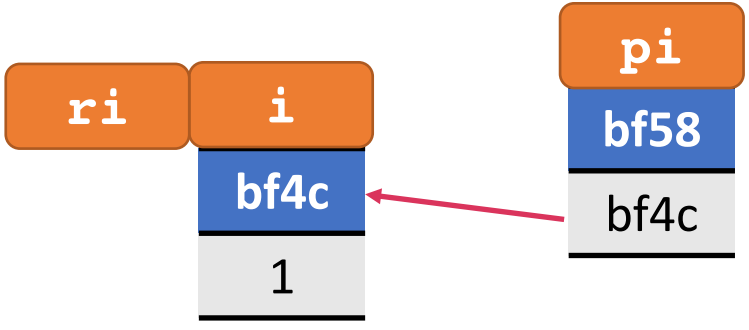
Pointers and references in C++

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
```



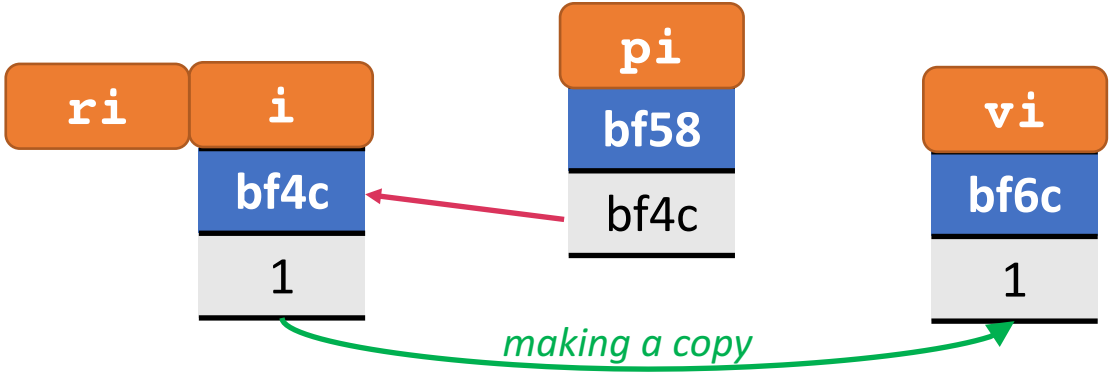
Pointers and references in C++

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
int* pi = &i; // pi is a pointer to i
```



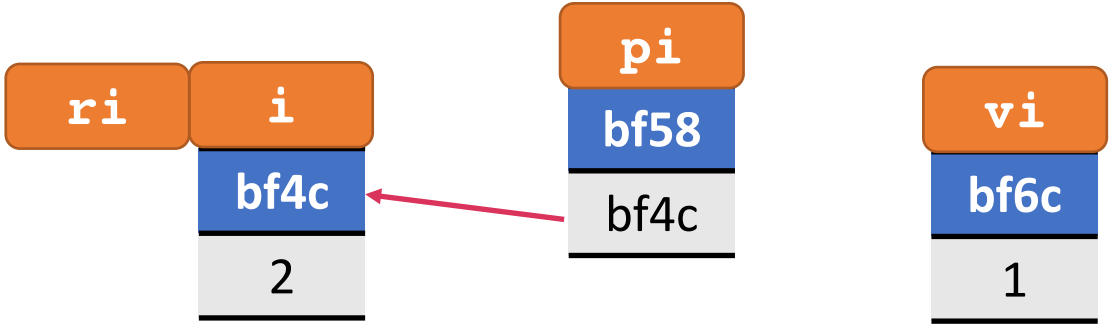
Pointers and references in C++

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
int* pi = &i; // pi is a pointer to i
int  vi = i;  // vi is a copy of i
```



Pointers and references in C++

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
int* pi = &i; // pi is a pointer to i
int  vi = i;  // vi is a copy of i
++i;          // increment i
```



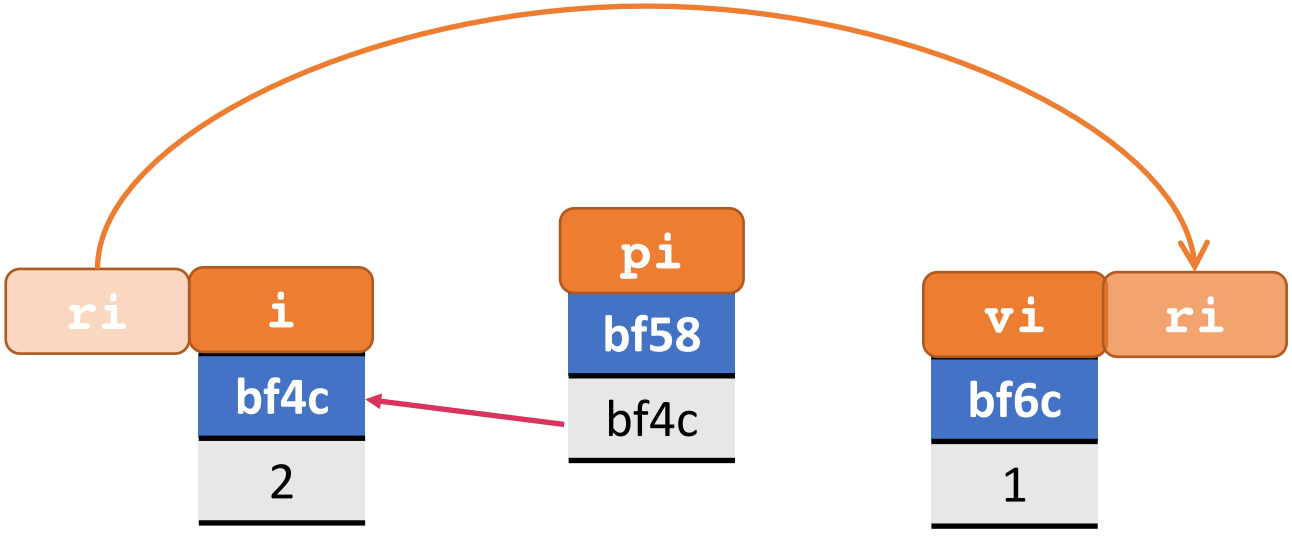
Example output

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
int* pi = &i; // pi is a pointer to i, *pi is value of i
int  vi = i;  // vi is a copy of i

++i;          // increment i

cout << i << "\t" << (void*)&i << endl;
cout << ri << "\t" << (void*)&ri << endl;
cout << *pi << "\t" << (void*)pi << endl;
cout << vi << "\t" << (void*)&vi << endl;
```

```
2 0x7fff5fbfbc14
2 0x7fff5fbfbc14
2 0x7fff5fbfbc14
1 0x7fff5fbfbc10
```

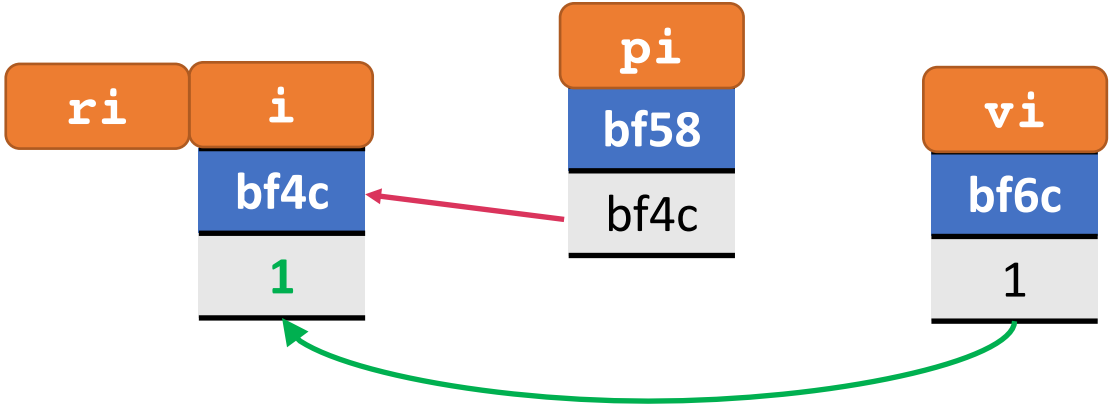
Can a reference to be "re-assigned?"

```
ri = vi;          // what will happen?
                  // will ri become
                  // an alias of another variable?
```



Or there is no "reassignment" but only "copy"?

```
ri = vi;          // what will happen?
                  // or will the value of vi be
                  // copied to the value of ri?
```



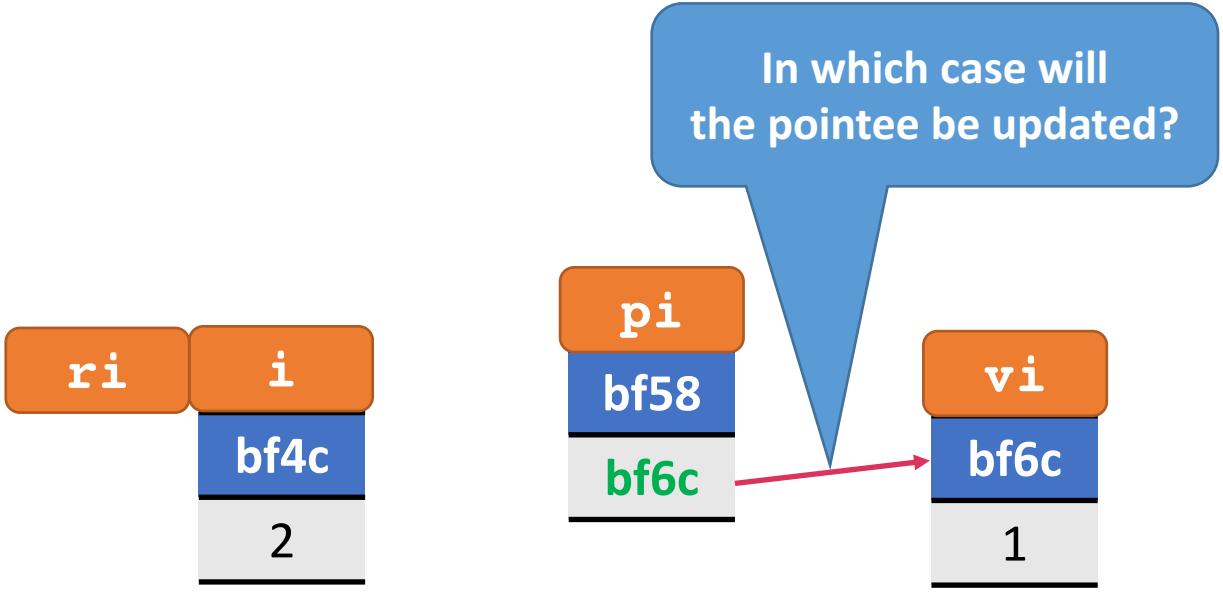
Which one was correct indeed?

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
int* pi = &i; // pi is a pointer to i
int  vi = i;  // vi is a copy of i

++i;          // increment i
ri = vi;      // what will happen?

cout << i << "\t" << (void*)&i << endl;
cout << ri << "\t" << (void*)&ri << endl;
cout << *pi << "\t" << (void*)pi << endl;
cout << vi << "\t" << (void*)&vi << endl;
```

```
1 0x7fff5fbfbc14
1 0x7fff5fbfbc14
1 0x7fff5fbfbc14
1 0x7fff5fbfbc10
```

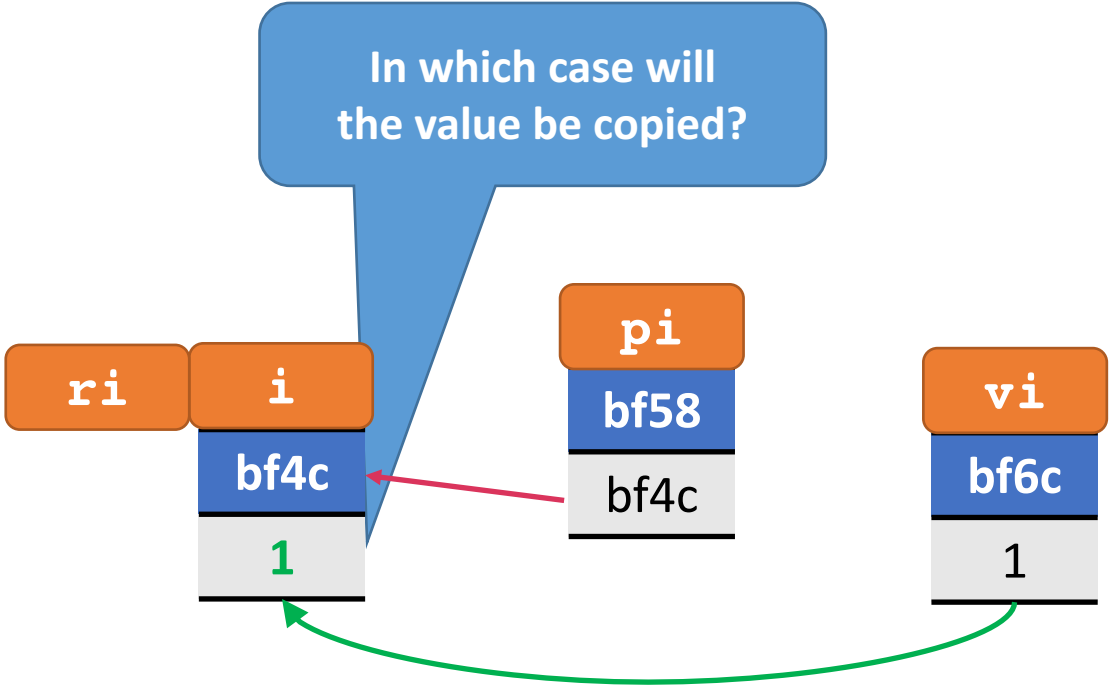
How about pointers? Can a pointer change its pointee?

```
pi = &vi;        // what will happen?
*pi = vi;        // how about this?
```



How about pointers? Can a pointer change its pointee?

```
pi = &vi;        // what will happen?
*pi = vi;        // how about this?
```



Assigning address can modify the pointee

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
int* pi = &i; // pi is a pointer to i
int  vi = i;  // vi is a copy of i

++i;          // increment i
pi = &vi;     // what will happen?
++i;          // another increment

cout << i << "\t" << (void*)&i << endl;
cout << ri << "\t" << (void*)&ri << endl;
cout << *pi << "\t" << (void*)pi << endl;
cout << vi << "\t" << (void*)&vi << endl;
```

```
3 0x7fff5fbfbc14
3 0x7fff5fbfbc14
1 0x7fff5fbfbc10
1 0x7fff5fbfbc10
```

Dereferencing only makes a copy

```
int  i  = 1;  // i is an integer
int& ri = i;  // ri is a reference to i
int* pi = &i; // pi is a pointer to i
int  vi = i;  // vi is a copy of i

++i;          // increment i*
*pi = vi;     // what will happen?
++i;          // another increment

cout << i << "\t" << (void*)&i << endl;
cout << ri << "\t" << (void*)&ri << endl;
cout << *pi << "\t" << (void*)pi << endl;
cout << vi << "\t" << (void*)&vi << endl;
```

```
2 0x7fff5fbfbc14
2 0x7fff5fbfbc14
2 0x7fff5fbfbc14
1 0x7fff5fbfbc10
```

So far..

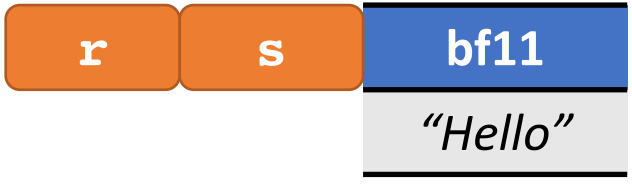
- **Default (value) type**
  - Stores a copy of an object
  - The copy could be “deep” or “shallow”.
    - Built-in and STL types are “deeply” copied
    - Pointers, and user-defined classes are usually “shallow” copied
- **Reference type**
  - Makes an “alias” of a value-type object
  - Somewhat easier to understand with limited usage
- **Pointer type**
  - Creates a pointer of an object
  - More flexible (allow `NULL`, pointee can be changed)

An STL object : std::string

```
string  s = "Hello"; // s is a STL string
```

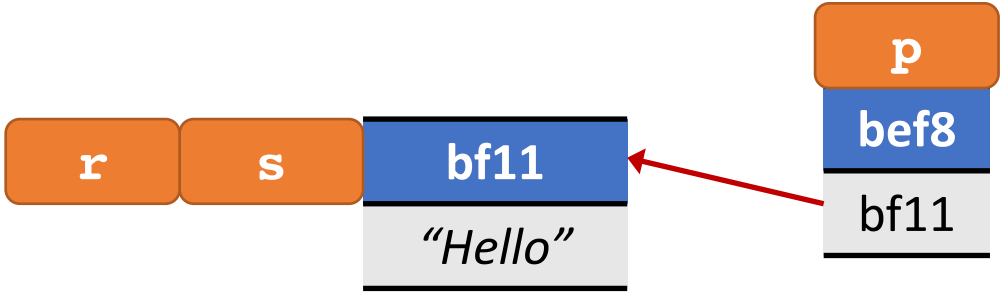
Making a reference of the string

```
string  s = "Hello"; // s is a STL string
string& r = s;       // r is a reference to s
```



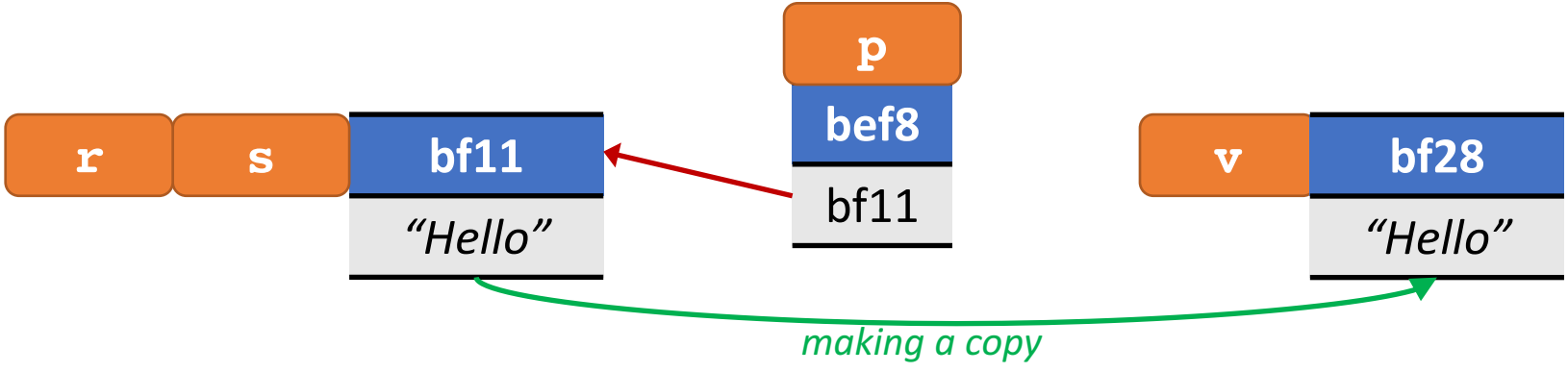
Creating a pointer of the string

```
string  s = "Hello"; // s is a STL string
string& r = s;       // r is a reference to s
string* p = &s;      // p is a pointer to s
```



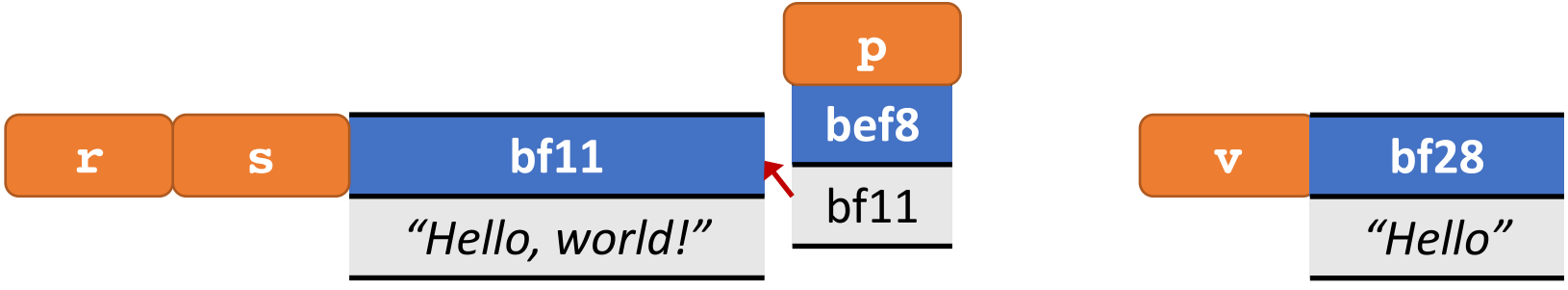
Making a (deep) copy of STL string

```
string  s = "Hello"; // s is a STL string
string& r = s;       // r is a reference to s
string* p = &s;      // p is a pointer to s
string  v = s;       // v is a copy of s
```



What happens upon modification?

```
string  s = "Hello"; // s is a STL string
string& r = s;       // r is a reference to s
string* p = &s;      // p is a pointer to s
string  v = s;       // v is a copy of s
s += ", world!";     // modify s.
```



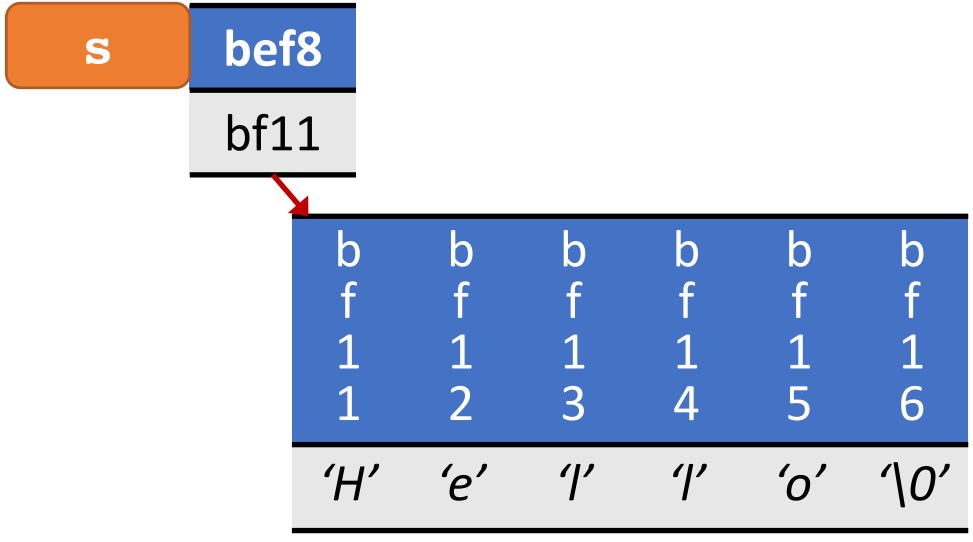
An example output

```
cout << s << "\t" << (void*)&s << endl;
cout << r << "\t" << (void*)&r << endl;
cout << *p << "\t" << (void*)p << "\t" << (void*)&p <<
endl;
cout << v << "\t" << (void*)&v << endl;
```

```
Hello, world! 0x7fff5fbfbc50
Hello, world! 0x7fff5fbfbc50
Hello, world! 0x7fff5fbfbc50 0x7fff5fbfbc48
Hello 0x7fff5fbfbc30
```

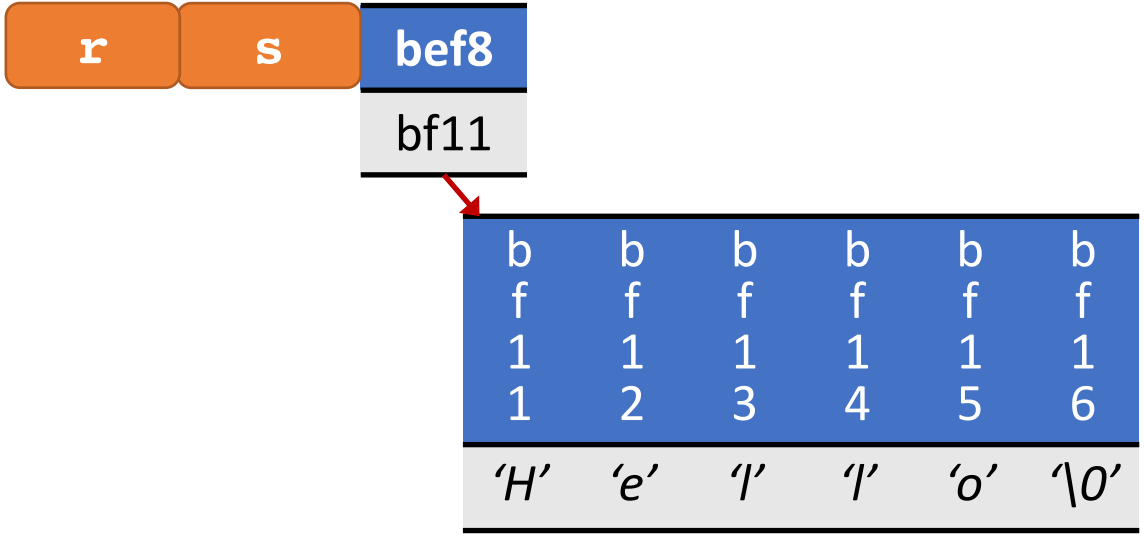
What happens to C-style strings?

```
const char* s = "Hello"; // s is a STL string
```



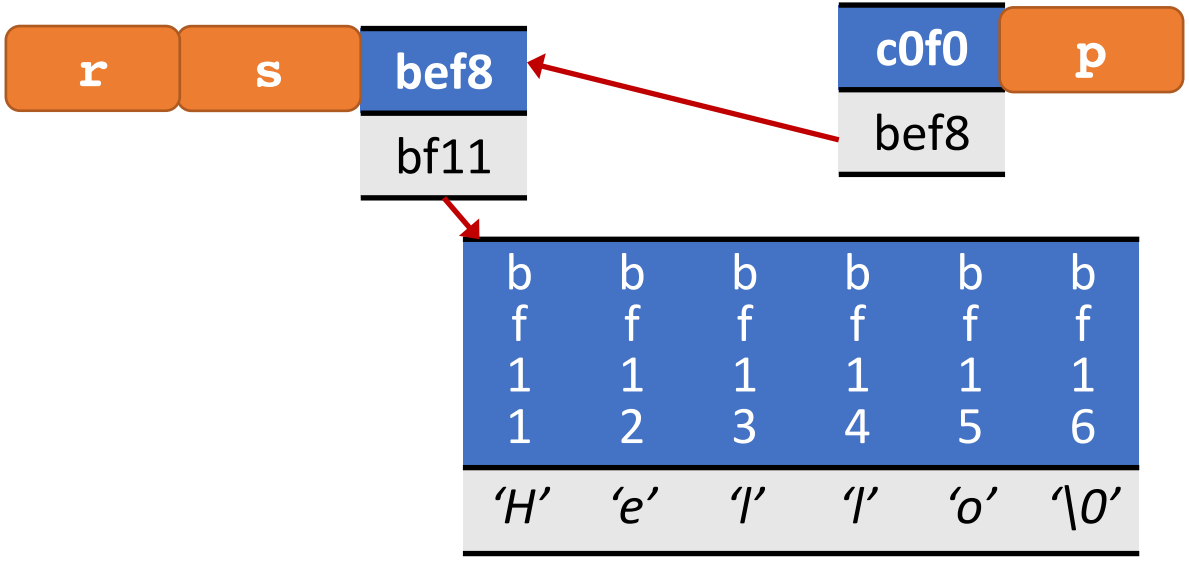
What happens to C-style strings?

```
const char* s = "Hello"; // s is a STL string
const char*& r = s;      // r is a reference to s
```



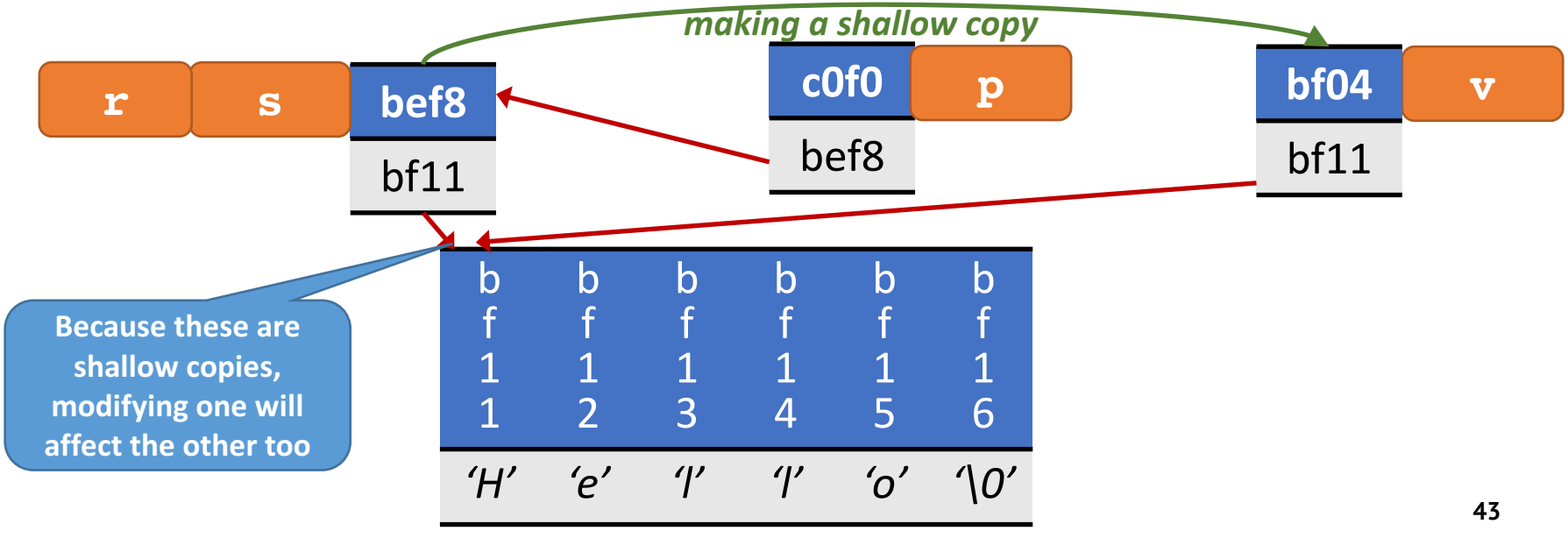
What happens to C-style strings?

```
const char* s = "Hello"; // s is a STL string
const char*& r = s;      // r is a reference to s
const char** p = &s;     // p is a pointer to s
```



What happens to C-style strings?

```
const char* s = "Hello"; // s is a STL string
const char*& r = s;      // r is a reference to s
const char** p = &s;     // p is a pointer to s
const char*  v = s;      // v is a copy of s
```



Reading Material

- [EK pp. 42-49] Using arrays and pointers
- [EK pp. 49-54] Functions