CS2311 Computer Programming

LT10: Pointer II

Computer Science, City University of Hong Kong

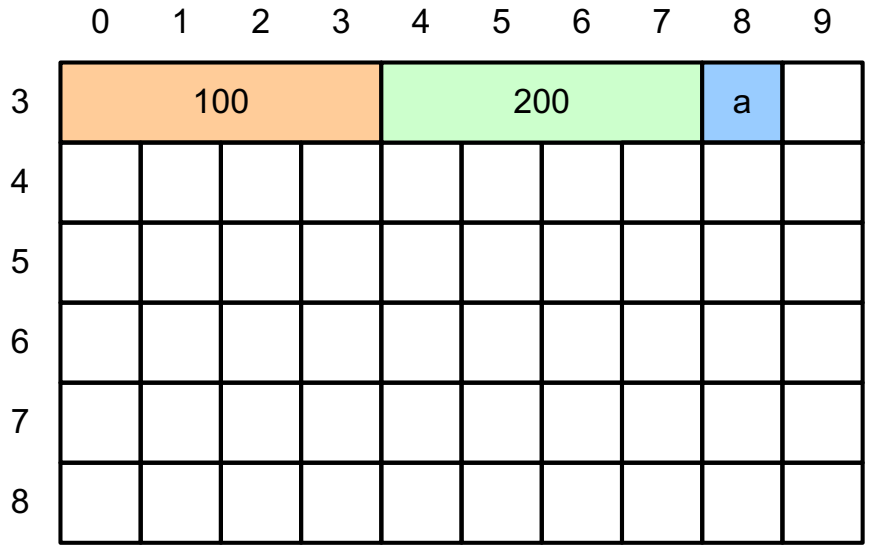
Semester B 2022-23

Review: Pointer 1

- Recap: variable and memory
- Pointer and its operations
- Pass by pointer
- Array and pointer

Review: Variable and Memory

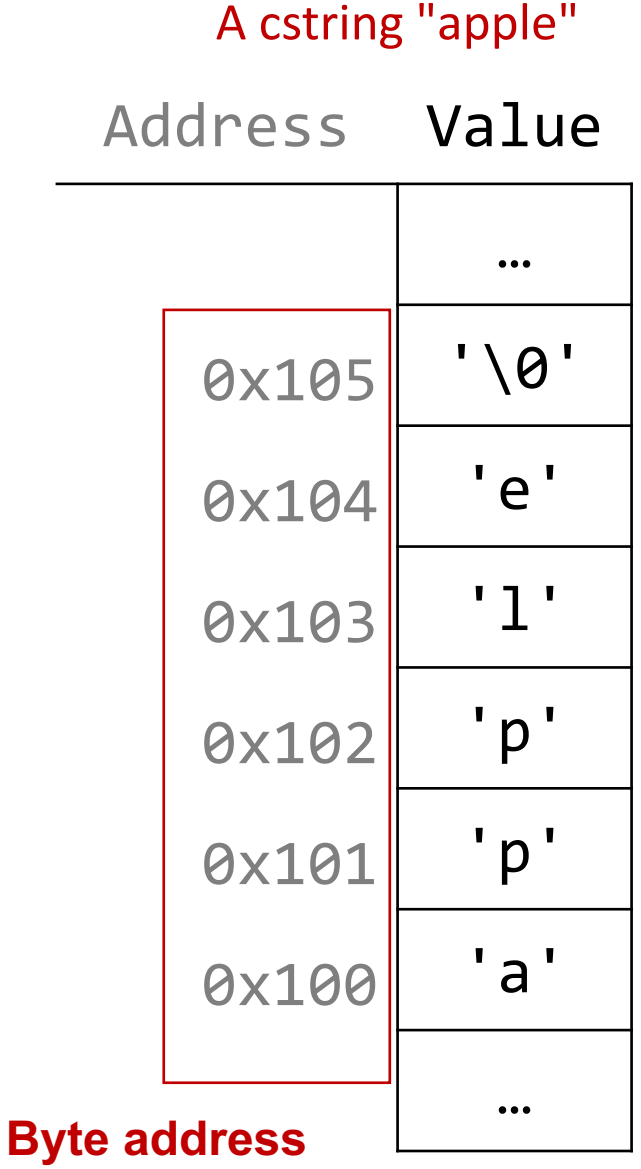
```
void main (){
        int  x;
        int  y;
        char c;
        x = 100;
        y = 200;
        c = 'a';
}
```



Identifier	Value	Address
x	100	30
y	200	34
c	‘a’	38

Review: Variable and Memory

- Most of the time, the computer allocates adjacent memory locations for variables declared one after the other
- A variable's address is the first **byte** occupied by the variable
- Address of a variable is usually in hexadecimal (base 16 with values 0-9 and A-F), e.g
  - 0x00023AF0 for 32-bit computers
  - 0x00006AF8072CBEFF for 64-bit computers



Review: What's a Pointer?

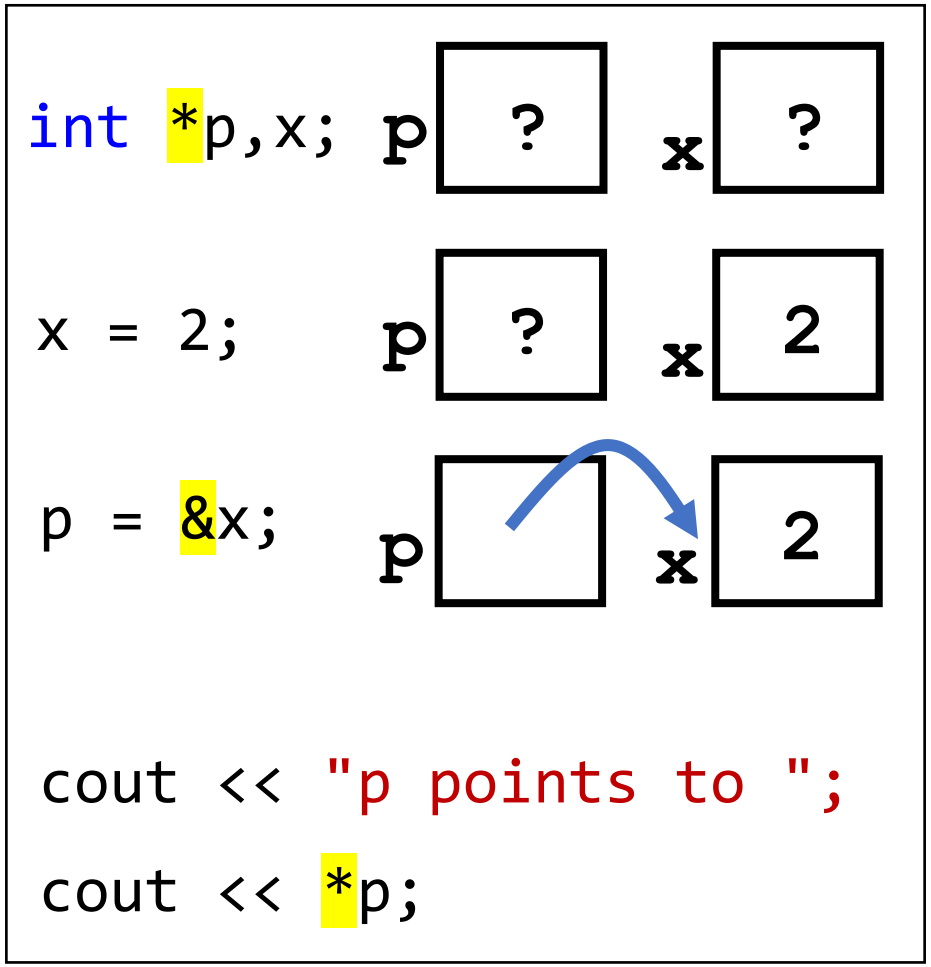
- Recall: data types
  - int, short, long: store the value of an integer
  - char: store the value of a character
  - float, double: store the value of a floating point
  - bool: store the value of a true or false
- Pointer is sort of another data type
  - Pointer store the value of a memory address
- A pointer is a variable which stores the memory address of another variable
- When a pointer stores the address of a variable, we say the pointer is pointing to the variable
- The pointer type is determined by the type of the variable it points to

Review: Basic Pointer Operators: & and *

& address operator: get address of a variable

***** is used in **TWO** different ways

- in declaration (such as int *p), it indicates a *pointer type* (e.g., int *p is a pointer which points to an int variable)
- when it appears in other statements (such as cout << *p), it's a *deference operator* which gets the value of the variable pointed by *p.*



Review: Common Pointer Operations

- Set a pointer *p1* point to a variable *x*

  p1 = &x;

- Set a pointer *p2* point to the variable pointed by another pointer *p1*

  p2 = *p1*; *// p2 and p1 now points to the same memory area*

- Update the value of the variable pointed by a pointer

  **p2* = 10;

- Retrieve the value of the variable pointed by a pointer

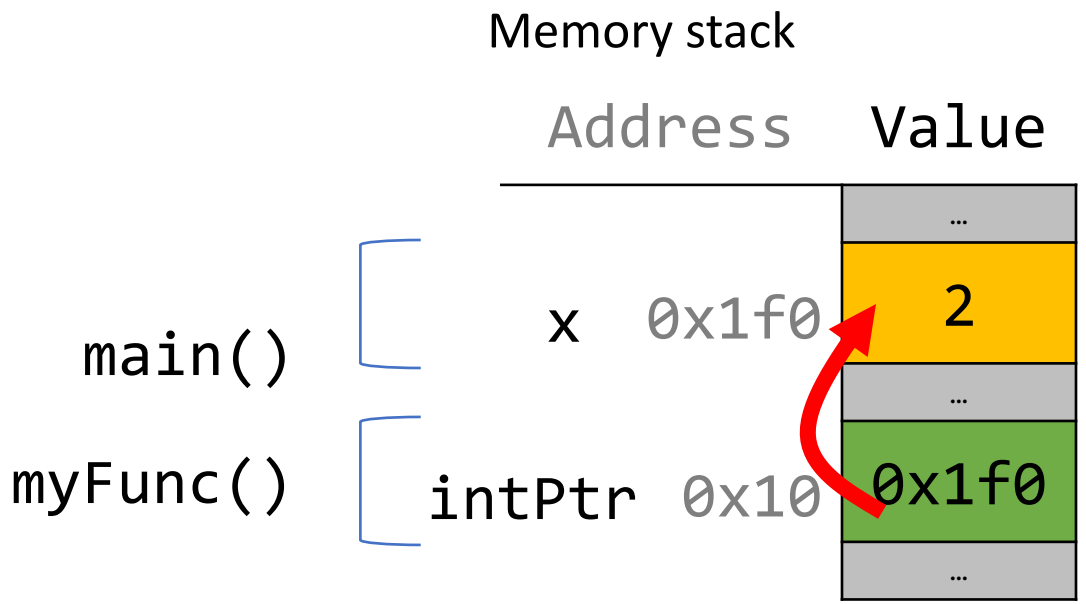
  int x = **p2*;

Review: Common Errors

```
int x = 3;

char c = 'a';

char *ptr;

ptr = &x;  // error: ptr can only points to a char, not int

ptr = c;   // error: cannot assign a char to a pointer

           // A pointer can only store a memory address

ptr = &c;
```

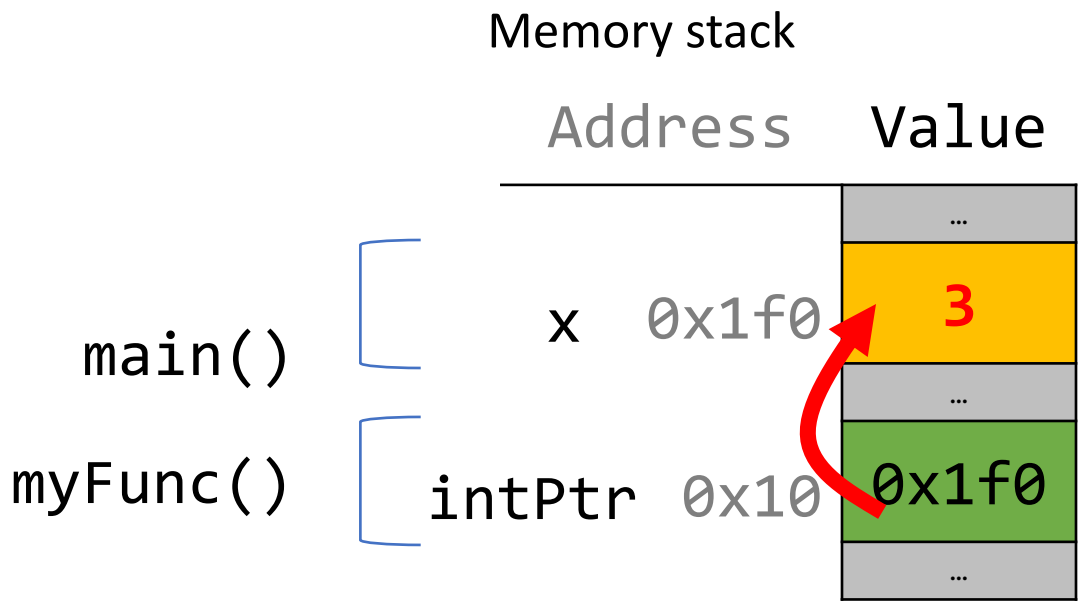
Review: Pass-by-Pointer

```
void myFunc(int* intPtr) {
    *intPtr = 3;
}

int main() {
    int x = 2;
    myFunc(&x);
    cout << x; // 3!
    return 0;
}
```



Review: Pass-by-Pointer

```
void myFunc(int* intPtr) {
     *intPtr = 3;
}

int main() {
     int x = 2;
     myFunc(&x);
     cout << x; // 3!
     return 0;
}
```



Pass-by-Pointer

- If you are performing an operation with some input and do not care about any changes to the input, **pass-by-value**. This makes a copy of the data.
- If you are modifying a specific instance of some value, **pass-by-reference** or **pass-by-pointer** of what you would like to modify. This makes a copy of the data's address.
- pass-by-pointer is more **efficient** and **powerful** than pass-by-value
  - gives the called function a *key* to open the door of the caller's memory
- on the other side of the coin: pass-by-value is **safer**

Review: Pointer 1

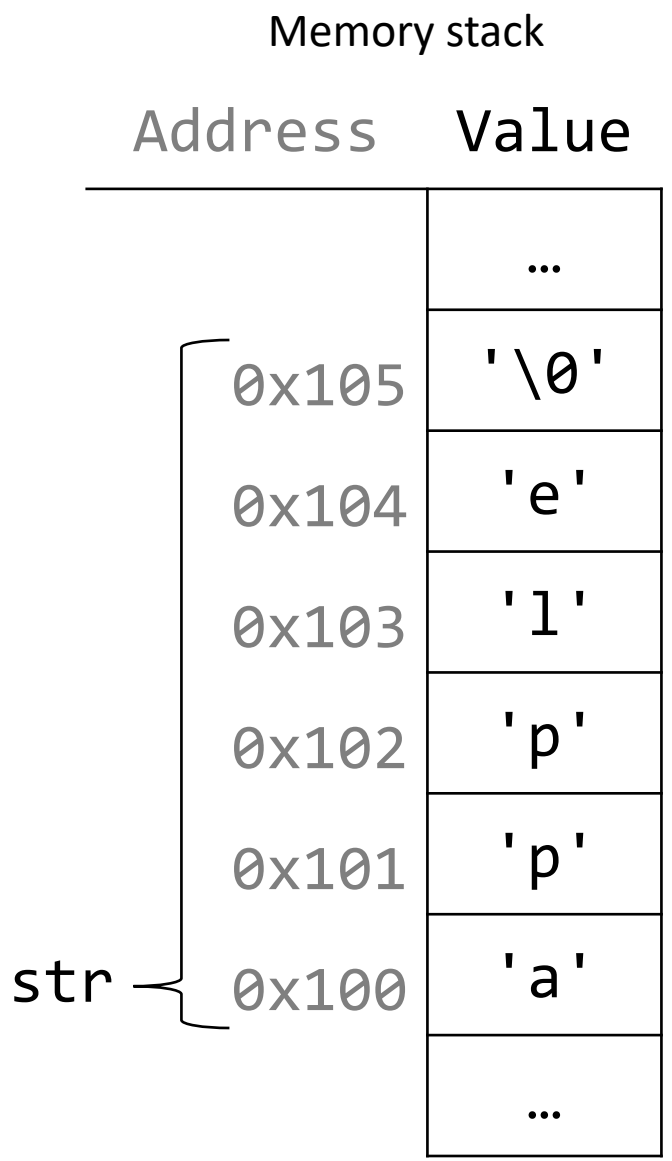
- Memory and variable
- Pointer and its operations
- Pass by pointer
- Array and pointer

Array Variable

- when we declare an array of characters, continuous memory is allocated on the memory stack to store the contents of the entire array

```
char str[6];
strcpy(str, "apple");
cout << str;
```

- the array variable (e.g. **str**) refers to the address of the first array element



char *

- A char * is technically a pointer to a **single character**.

- We can use char * as a string (cstring), which starts from the character it points to until the null terminator.

```
char str[] = "Hello World";

char *p = &str[0]; cout << p << endl; // "Hello World"

     p = &str[3]; cout << p << endl; // "lo World"
```

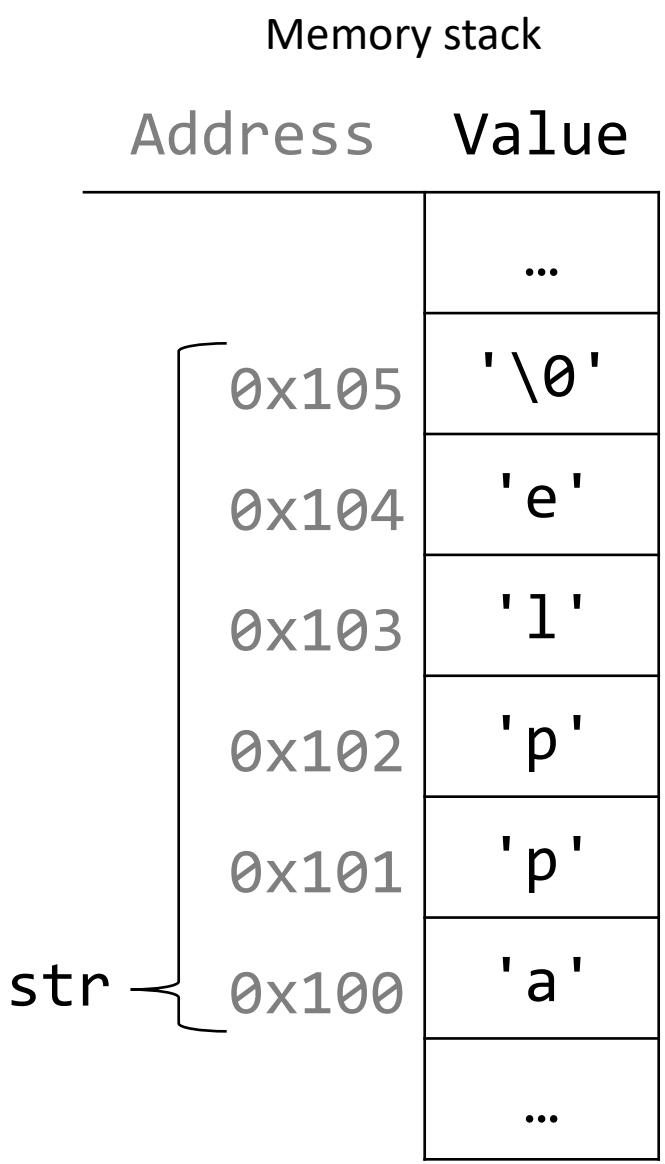
Array Variable is NOT a Pointer

- when we declare an array of characters, continuous memory is allocated on the memory stack to store the contents of the entire array

```
char str[6];
strcpy(str, "apple");
cout << str;
```

- the array variable (e.g. **str**) refers to the address of the first array element, but str is not a pointer!
- For example, sizeof(str) returns the size of the array but sizeof a pointer returns address length

```
cout << sizeof(str) << "\n"; // 6
cout << sizeof(&str[0]); // 8 or 4
```



Array Variable is NOT a Pointer

- Reassignment of array variable is NOT allowed

```
char str1[] = "Hello";
char str2[] = "World";
str1 = str2; // NOT allowed
```

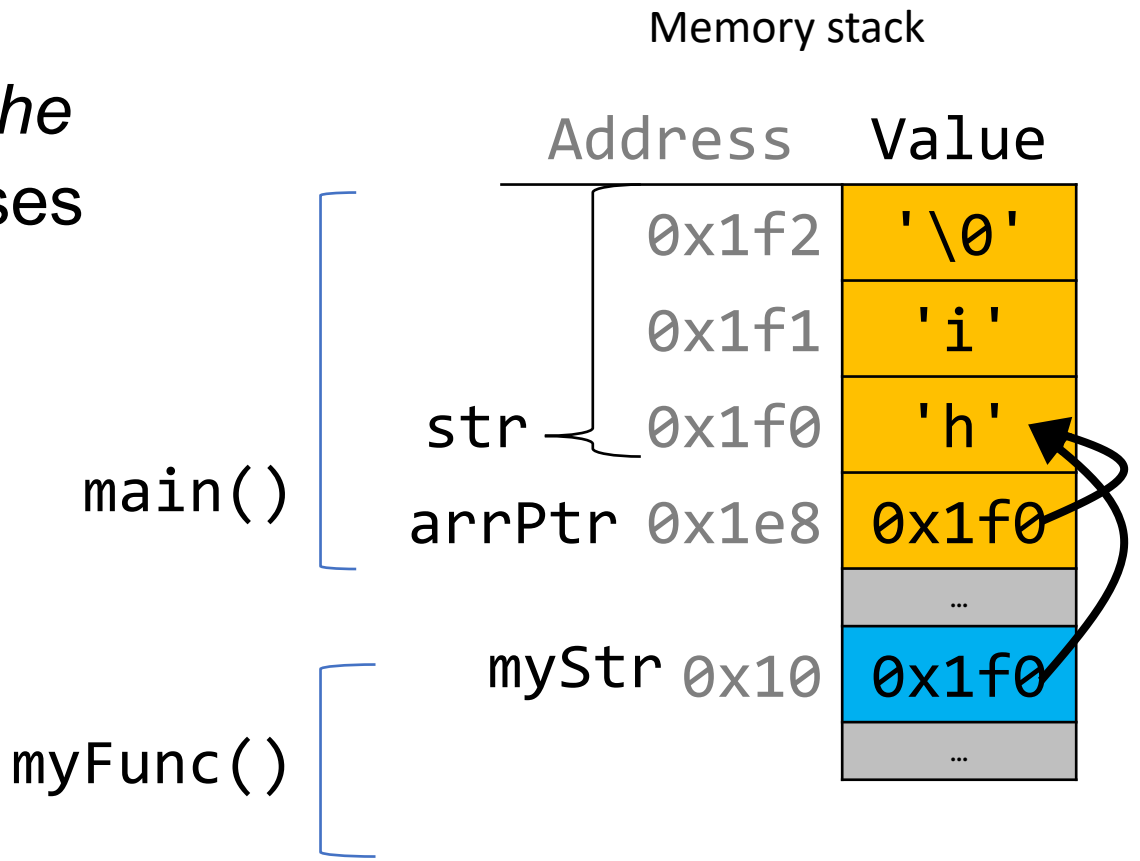
- In comparison, reassignment of pointer is allowed

```
char  str1[] = "Hello";
char  str2[] = "World";
char *ptr  = str1; cout << ptr << "  ";
      ptr  = str2; cout << ptr << "\n";
```

Arrays as Parameters

- when you pass an **array** variable as a pointer-type parameter, making a *copy of the address of the first array element and* passes it as a pointer to the function.

```
void myFunc(char *myStr) {
      ...
}
void main() {
      char str[3];
      strcpy(str, "hi");
      // equivalent myFunc(str);
      char *arrPtr = str;
      myFunc(arrPtr);
}
```



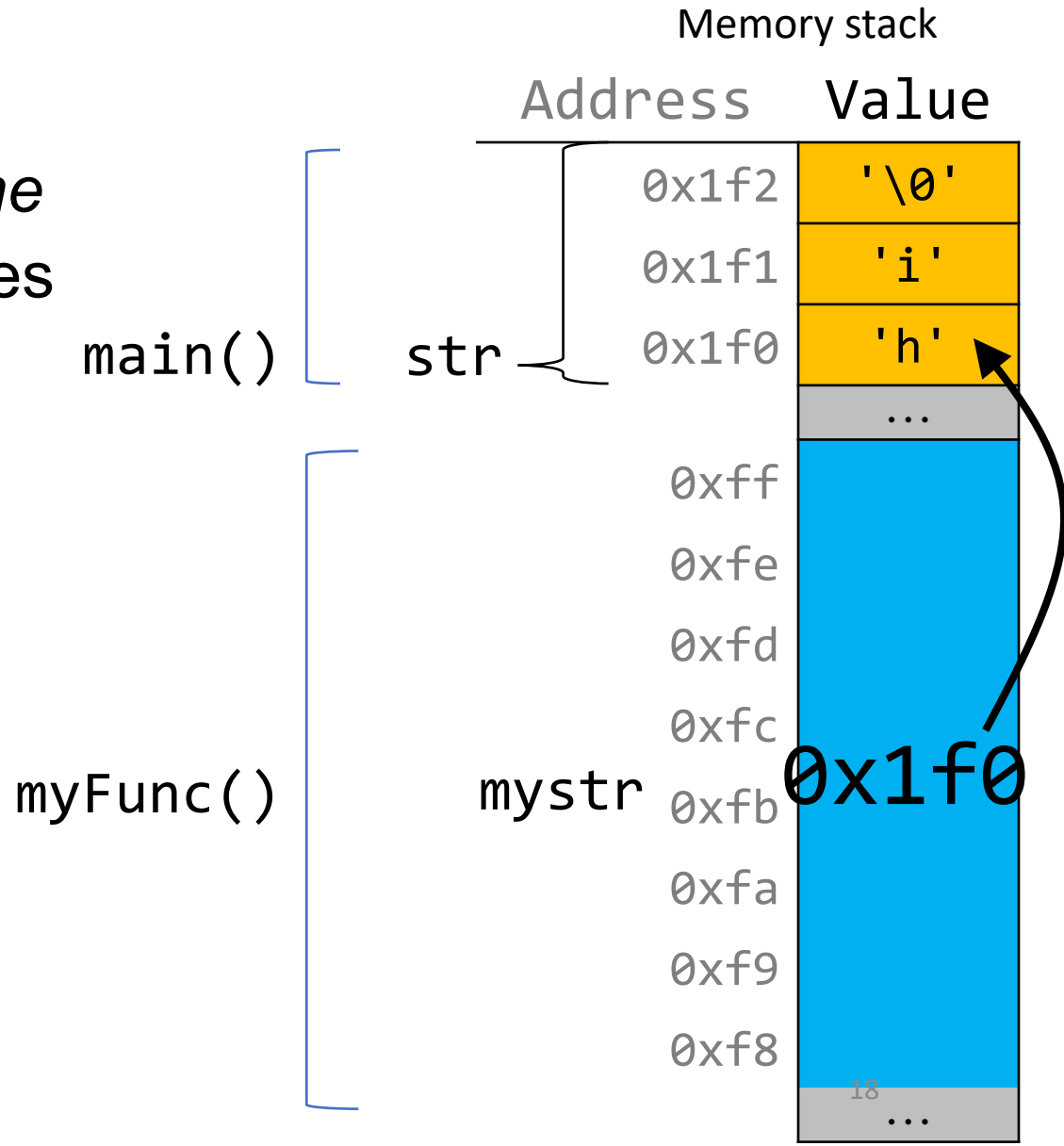
Calling with myFunc(arrPtr)

Arrays as Parameters

- when you pass an **array** variable as a pointer-type parameter, making a *copy of the address of the first array element and* passes it as a pointer to the function.

```
void myFunc(char *myStr) {
      ...
}
void main() {
      char str[3];
      strcpy(str, "hi");
      // equivalent myFunc(str);
      char *arrPtr = str;
      myFunc(arrPtr);
}
```

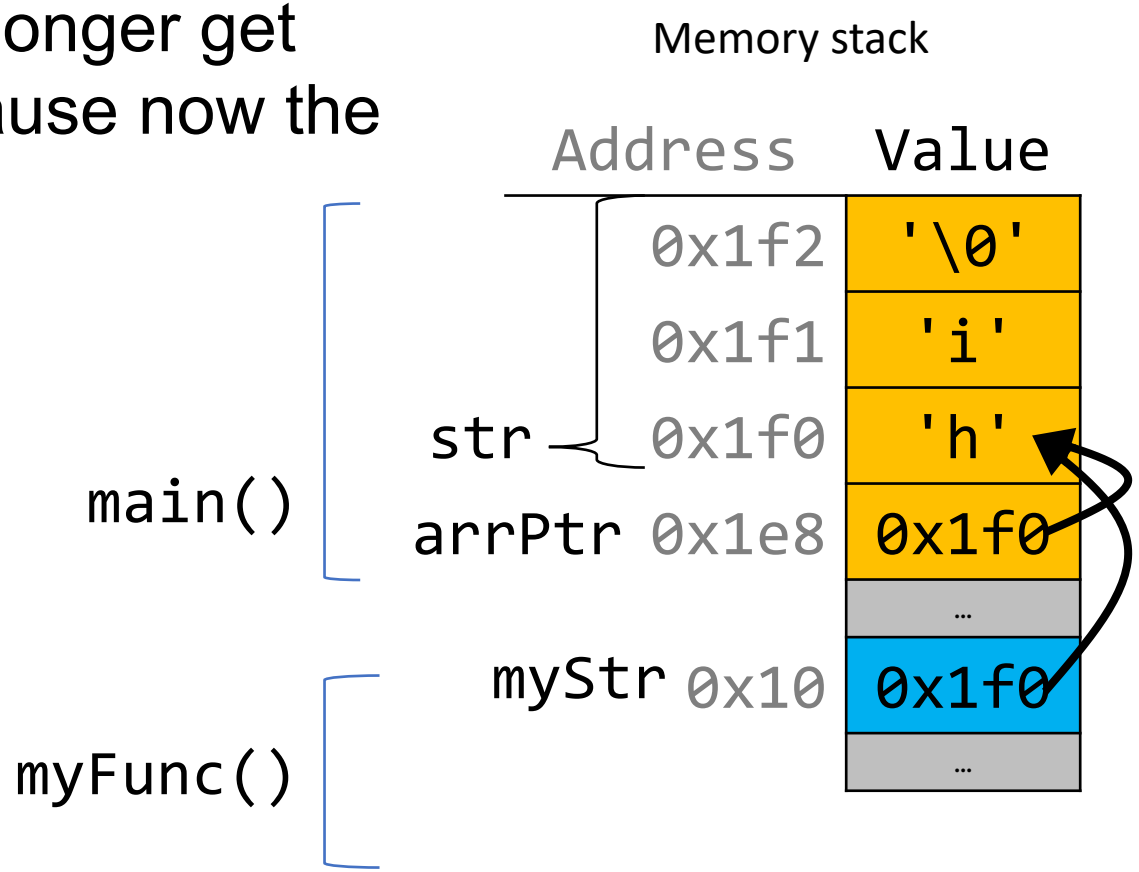
Calling with myFunc(str)



Arrays as Parameters

- however, with pass-by-pointer, we can no longer get the full size of the array using **sizeof**, because now the array variable is passed as a pointer,

```
void myFunc(char *myStr) {
        cout << sizeof(myStr); // 4 or 8
 }
 void main() {
        char str[3];
        strcpy(str, "hi");

        cout << sizeof(str); // 3
        myFunc(str);
 }
```



Arrays as Parameters

- All string functions take char * parameters – they accept char[], but they are implicitly converted to char * before being passed.
  - strlen(char *str); strcmp(char *str1, char *str2) …
- char * is still a string in all the core ways a char[] is
  - Access/modify characters using bracket notation
  - Use string functions
  - print
- But under the hood they are represented differently!
- **Takeaway:** We create strings as char[], pass them around as char *

Arrays vs Pointers Summary

- When you create an array, you are making space (allocate memory) for each element in the array.
- When you create a pointer, you are making space for a 4 or 8 byte address.
- Arrays “decay to pointers” when you pass as parameters.
- You cannot set an array equal to something after initialization, but you can set a pointer equal to something at any time.
- &arr does nothing on arrays, but &ptr on pointers gets its address
- sizeof(arr) gets the size of an array in bytes, but sizeof(ptr) is always 4 or 8

Exercise 1

```
// Assume user input 4 3 2 1 0. What's the output and why?
int main() {
	int a=1, c[4], b=1, i=0;
	while (true) {
		cin >> c[i];
		if (c[i] == 0)
			break;
		else
			i++;
	}
	cout << a+b << endl;
	return 0;
}
```

Exercise 2

What's the output of the following codes?

```
int x=20, y=30;
int *p = &x;
int *q = &y;
cout <<  *p  << " "  << *q  << endl;
for (int i=0; i<2; i++)
    *p += *q + 3;
cout  <<  *p  <<  " "  <<  *q  << endl;
cout <<  x  <<  " "  <<  y  << endl;
```

Exercise 3

What's the output of the following codes?

```
char b[100] = "Hello World";
char *buffer = &b[1];
strcpy(buffer, "World");
cout << b << endl;
cout << buffer << " " << sizeof(buffer) << endl;
```

Outline Today

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Pointer Arithmetic

- You can perform arithmetic operations on a pointer with four operators
  - ++, --, +, and –
- When you do arithmetic with a pointer *p*, you consider *p* points to an array, and you perform arithmetic as it's an array index
- e.g.

```
int a[4] = {0, 1, 2, 3};
int *p = &a[3];
p -= 2; // now p points to a[1]
cout << *p << endl;
p++;    // now p points to a[2]
cout << *p << endl;
```

Pointer Arithmetic

Byte address, which means, 4 bytes

```
1  int a[6] = {0, 1, 2, 3, 4, 5}; // assume a[0] stored in 0x16d5bf730
2  int *pa = &a[1];
3  cout << hex << pa   << endl; // output "0x16d5bf734"
4  cout << hex << ++pa << endl; // output "0x16d5bf738"
5  cout << &a[5]-pa    << endl; // output "3"
6
7  long b[6] = {5, 4, 3, 2, 1, 0}; // assume b[0] stored in 0x16d5bf700
8  long *pb = &b[1];
9  cout << hex << pb   << endl;
10 cout << hex << ++pb << endl;
11 cout << &b[4]-pb    << endl;
```

```
0x16d5bf708
0x16d5bf710
2
```

Pointer Arithmetic

- Pointer arithmetic is equivalent to array index arithmetic

```
1  char str[] = "Hello World";
2  char *p = str;
3  cout << p+6      << endl;
4  cout << &str[6] << endl;
5  cout << str[2]  << endl;
6  cout << *(p+2)  << endl;
```

```
World
World
l
l
```

Pointer Arithmetic: common errors

- Multiplication and division of pointers are not allowed in C++

```
int *ptr1, *ptr2, *ptr3;

ptr3 = ptr1 * ptr2; // Error: Multiplication of pointers

ptr3 = ptr1 / ptr2; // Error: Division of pointers

int a = 1, b = 2, c = 3;

*ptr1 = &a; *ptr2 = &b; *ptr3 = &c;

*ptr3 = *ptr1 * *ptr2; // No error: c = a * b

*ptr3 = *ptr1 / *ptr2; // No error: c = a / b
```

Pointer Arithmetic Summary

Equivalent representation		Remark
`num`	`&num[0]`	num is the address of the 0th element of the array
`num+i`	`&(num[i])`	Address of the i-th element of the array
`*num`	`num[0]`	The value of the 0-th element of the array
`*(num+i)`	`num[i]`	The value of the i-th element of the array
`(*num)+i`	`num[0]+i`	The value of the 0-th element of the array plus *i*

Exercise 1

- What's the output of the following program

```
#define N 10

int main() {
    int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum = 0;
    for (int i = 0; i < N; ++i)
        sum += *(a+i);
    cout << sum;
    return 0;
}
```

Exercise 2

- Suppose we use a variable str as follows

```
str = str+1;
str[1] = 'a';
cout << str;
```

- For each of the initializations on the right,
  - will there be a compilation error/runtime error?
  - if no error, what's the output

2: ealo3
4: ealo4

```
1. char str[7];
   strcpy(str, "Hello1");

2. char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;

3. char *str = "Hello2";

4. char ptr[] = "Hello4";
   char *str = ptr;
```

Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Pointer Array

- A pointer array's elements are all pointers.
- For example,

```
int a[6] = {0,1,2,3,4,5};
int *m[2] = {&a[0], &a[3]};
for (int row=0; row<2; row++) {
      for (int col=0; col<3; col++)
            cout << m[row][col] << " ";
      cout << "\n";
}
```

```
0 1 2
3 4 5
```

Pointer Array

- int main(int argc, char *argv[])
- Allows main to take parameter from user input
- int argc: number of arguments to take
- char *argv[]: array of arguments, each is a string

Pointer Array

```
// ./main apple banana orange peach pear

#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
      cout << "Have " << argc << " arguments: " << endl;
      for (int i = 0; i < argc; i++)
            cout << argv[i] << endl;
      return 0;
}
```

```
Have 6 arguments:
./main
apple
banana
orange
peach
pear
```

Array Pointer

- Pointer to a one-dimensional array can be declared as:

```
int arr[] = {1,2,3,4,5};
int *p; p = arr;
```

- Similarly, pointer to a two-dimensional array can be declared as:

```
int arr[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
int *p[3] = arr;    // cannot declare as an array of 3 pointers
```

Array Pointer

- Pointer to a one-dimensional array can be declared as:

```
int arr[] = {1,2,3,4,5};
int *p; p = arr;
```

- Similarly, pointer to a two-dimensional array can be declared as:

```
int arr[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
int *p[3] = arr;    // cannot declare as an array of two pointers
int (*p)[3] = arr; // a pointer to an array of three integers
cout << *(*(p+1)+2) << endl; // p[1][2]
cout << *(p[2]+1)   << endl; // p[2][1]
```

Pass 2D Array to Function

```
void foo(int x[][10]) { // the size of the second dimension MUST be given
    …                   // the size of the first dimension is optional
}
void main() {
    int y[20][10];
    foo(y);
}
```

Pass Array Pointer to Function

```
void foo(int (*x)[10]) {  //  pointer to an array of 10 integers
    …
}
void main() {
    int y[20][10];
    foo(y);
}
```

Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & Pointer reference
- Dynamic memory allocation

Pointer of Pointer

- Example:

```
int a = 4;
int *p = &a;
int **pp = &p; //  pp is a pointer to an int pointer
cout << *p << endl;
cout << **pp << endl;
```

Pointer of Pointer

- Example:

```
int a = 4;
int *p = &a;
int **pp = &p; // pp is a pointer to an int pointer
cout << *p << endl;
cout << **pp << endl;
cout << hex << p << endl;
cout << hex << pp << endl;
cout << hex << *pp << endl;
```

```
4
4
0x16dddf754
0x16dddf748
0x16dddf754
```

Why Need Pointer of Pointer?

- Example: write a program to skip leading spaces in a string

- Does the right-side program work? Why?

hello
  hello

```
void skipSpaces(char *p) {
        while (*p == ' ')
                p++;
        cout << p << endl;
}

int main() {
        char str[] = "    hello";
        char *p = str;
        skipSpaces(p);
        cout << p << endl;
        return 0;
}
```

Why Need Pointer of Pointer?

- Example: write a program to skip leading spaces in a string

- We want the called function to modify the pointer, so …

```
hello
hello
```

```
void skipSpaces(char **p) {
        while (**p == ' ')
                (*p)++;
        cout << *p << endl;
}

int main() {
        char str[] = "    hello";
        char *p = str;
        skipSpaces(&p);
        cout << p << endl;
        return 0;
}
```

Pointer's Pointer vs Pointer Reference

```
void skipSpaces(char **p) {
	while (**p == ' ')
		(*p)++;
	cout << *p << endl;
}
int main() {
	char str[] = "    hello";
	char *p = str;
	skipSpaces(&p);
	cout << p << endl;
	return 0;
}
```

```
void skipSpaces(char* &p) {
	while (*p == ' ')
		p++;
	cout << p << endl;
}
int main() {
	char str[] = "    hello";
	char *p = str;
	skipSpaces(p);
	cout << p << endl;
	return 0;
}
```

Quick Summary

- Array of pointer

```
int *a[2];
```

- Pointer of array

```
int a[4][2] = {{0,1}, {2,3}, {4,5}, {6,7}}; int (*p)[2] = a;
cout << p[2][1] << " " << *(*(p+2)+1) << " " << *(p[2]+1);
```

- Pointer of pointer

```
int a=4; int *p=&a; int **pp=&p; cout << **pp;
```

- Pointer reference

```
void func(char* &p);
```

Exercise 1

- What's the output of the following codes?

```
int arr[4][3] = {{1,2,3}, {4,5,6}, {7,8,9}, {10,11,12}};
int (*p)[3] = arr;



cout << *(*(p+2)+1) << endl;
cout << *(p[3]+2)   << endl;
```

Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & pointer reference
- Dynamic memory allocation

Motivation

- In C/C++, the size of a statically allocated array has a limit

```
const unsigned int size = 0xffffffff;
int a[size];
```

- Sometime, we need to determine the array size at runtime

```
int size;
cin >> size;
int a[size];
```

Dynamic Memory Allocation

- Dynamic memory: memory that can be *allocated*, *resized*, and *freed* during program runtime.
- When do we need dynamic memory?
  1. when you need a very large array
  2. when we do not know how much amount of memory would be needed for the program beforehand.
  3. when you want to use your memory space more efficiently.
     - e.g., if you have allocated memory space for a 1D array as array[20] and you end up using only 10 memory

Dynamic Memory Allocation

- Keywords: new & delete

```
// Declaration
int *p0 = new int(10); // init an integer 10 in memory, make p0 point to it
char *p1 = new char('a'); // init a char 'a' in memory, make p1 point to it

// Free memory is your duty. Otherwise, the memory space cannot be reused
delete p0; // free the memory pointed by p0
delete p1; // free the memory pointed by p1

// Will be illegal after deletion
*p0 = 10;
```

Dynamic Memory Allocation

- Syntax on array: new [] and delete []

```
//  Declaration
int n; cin >> n;
int *p0 = new int[n]; //  allocate memory for an int array of n elements
char *p1 = new char[n]; //  allocate memory for a char array of n elements

// Free memory is your duty. Otherwise, the memory space cannot be reused
delete[] p0; //  free the memory pointed by p0
delete[] p1; //  free the memory pointed by p1
```

The NULL pointer

- A special value that can be assigned to **any** type of pointer variable
  - e.g., int *a = NULL; double *b = NULL;
- A **symbolic constant** defined in standard library headers, e.g. <iostream>
- When assigned to a pointer variable, that variable points to nothing
- Initialization after declaration

```
int *ptr1 = NULL;
```

- Check null pointer before using the pointer:

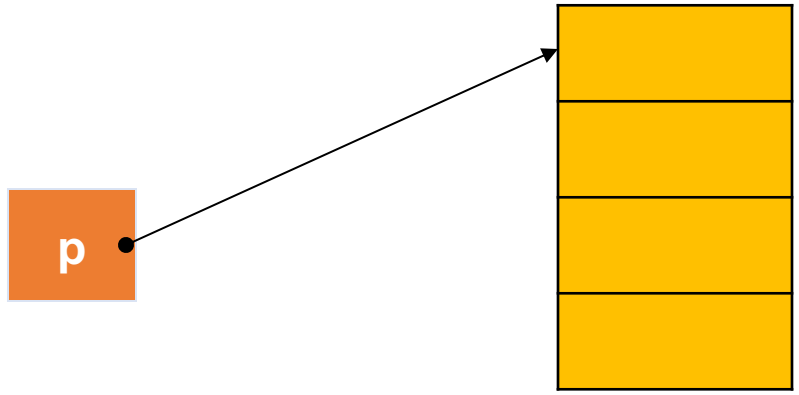
```
if (ptr)
if (!ptr)
```

Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```

Dynamic Memory Walkthrough

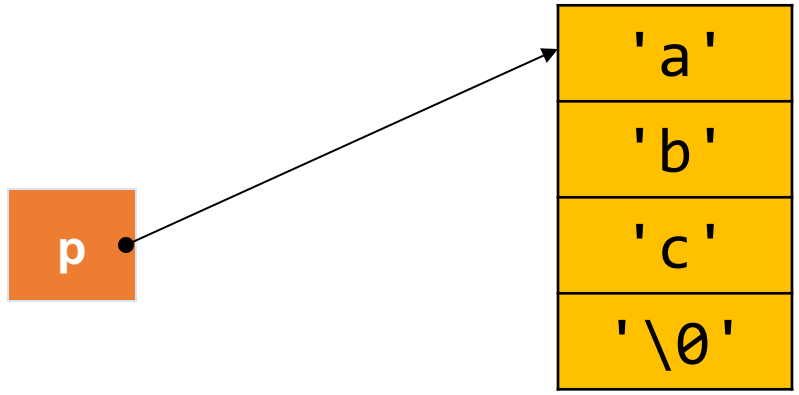
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



new dynamically allocates 4 bytes of memory. new returns a pointer to the 1st byte of the chunk of memory, which is assigned to s1

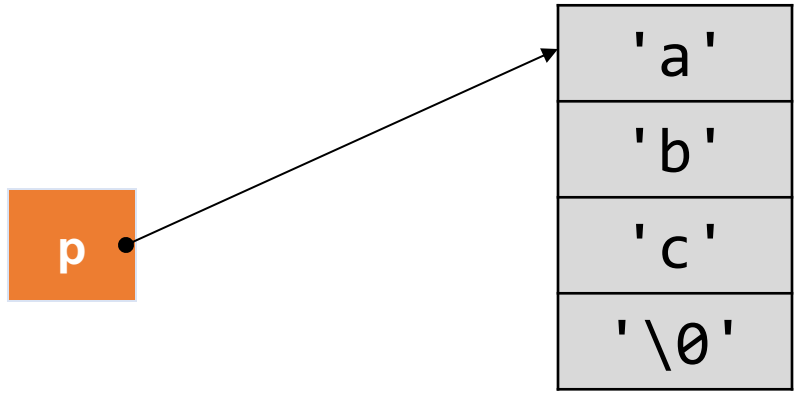
Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```

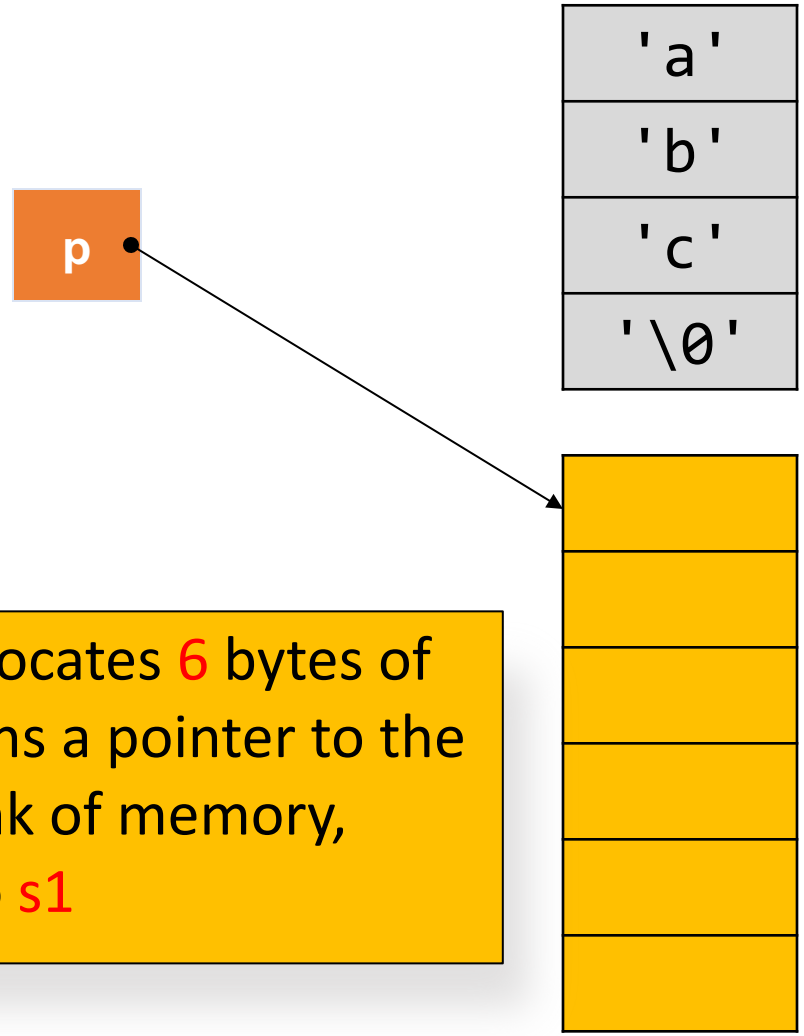


Grey memory means the block of memory is free and can be used to store other data.

p may or may not be pointing to the same address, and you can still print it, but that memory no longer belongs to p.

Dynamic Memory Walkthrough

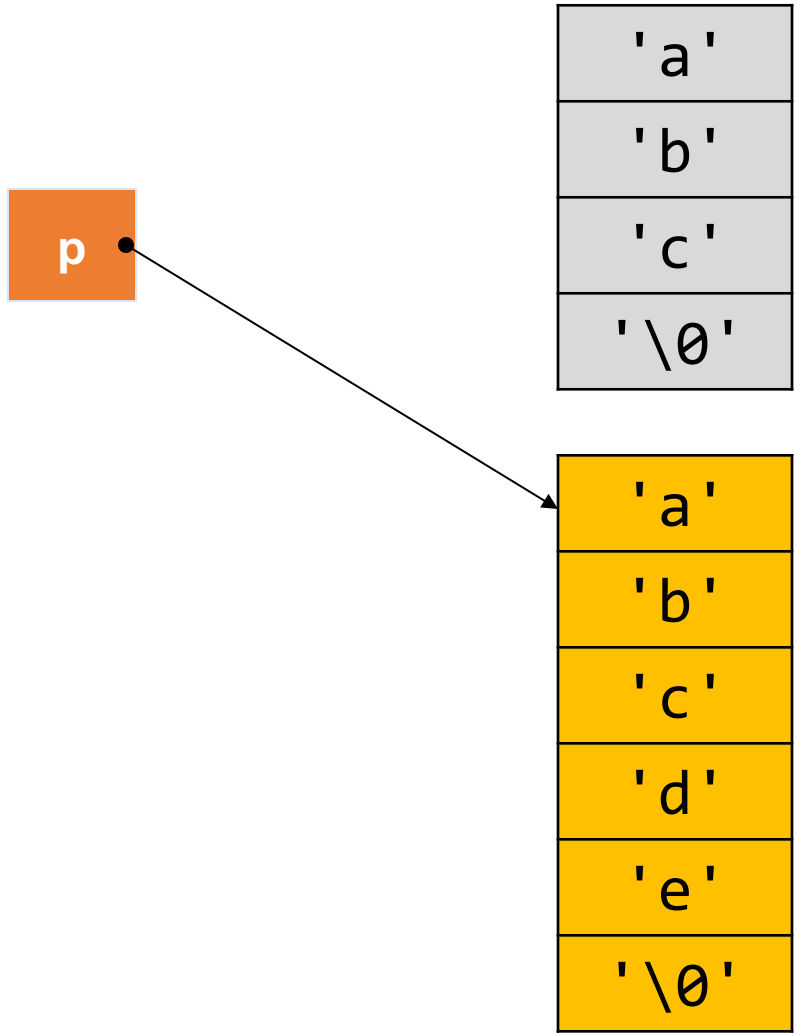
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



new dynamically allocates 6 bytes of memory. new returns a pointer to the 1st byte of the chunk of memory, which is assigned to s1

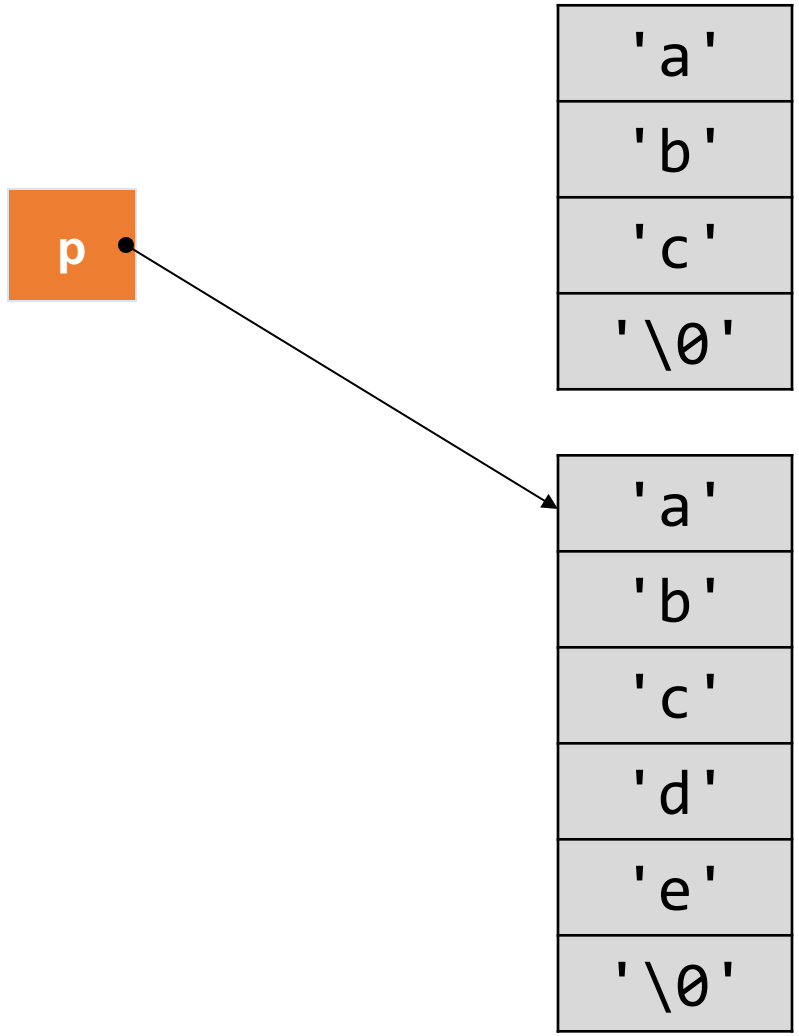
Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL;
```



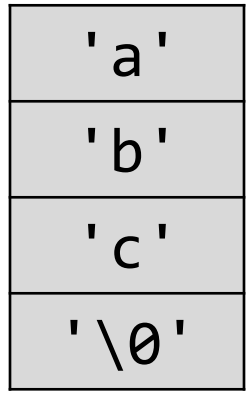
Dynamic Memory Walkthrough

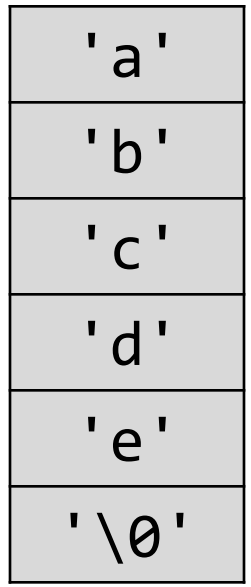
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL;
```



Dynamic Memory Walkthrough

```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1; // input "abcde"
cout << s1;
delete [] s1;
s1 = NULL; // optional
```





Exercise

- Write a function *readInput*() that can read all the integer inputs from the user and print out inputs in a reverse order.
- Assume the first input is n, indicating how many integers we will get from the user.

```
void readInput() {
      ...
}
```