CS2310 Computer Programming

LT10: Pointer II

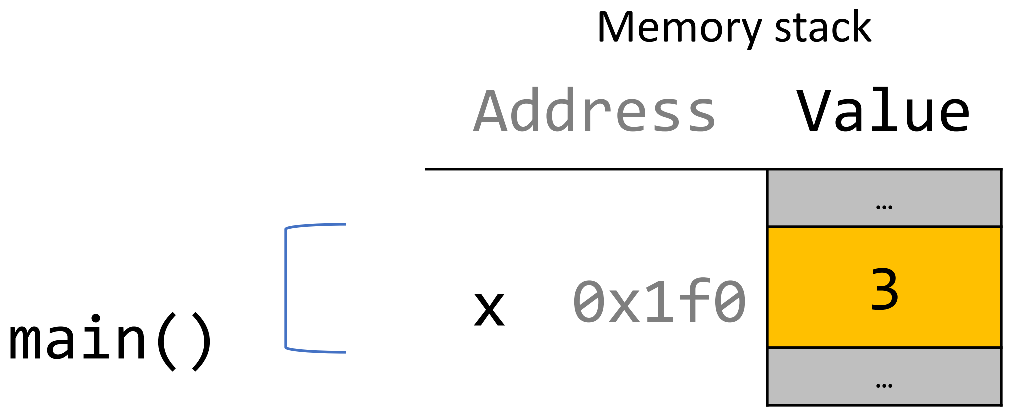
Computer Science, City University of Hong Kong

Semester A 2023-24

Recap: Pointer Operations

A pointer is a variable that stores a memory address.

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

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



Recap: Array Variable vs char *

```
char s0[] = "Hello";
char s1[] = "World";

char *p = &s0[0]; cout << p << endl;
      p = &s1[2]; cout << p << endl;

s0 = s1; // wrong: reassignment of array variable is not allowed

cout << sizeof(s0) << endl;
cout << sizeof(p)  << endl;
```

Recap: Array Variable vs char *

```
void foo(char str[]) {
	cout << "sizeof str in foo:   " << sizeof(str) << endl;
	cout << "length of str in foo:   " << strlen(str) << endl;
}

int main() {
	char str[] = "Hello World";
	cout << "sizeof str in main: " << sizeof(str) << endl;
	foo(str);
	return 0;
}
```

Outline

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

Pointer Arithmetic

When you do **pointer arithmetic**, you are adjusting the pointer by a certain *number of places*.

```
char *str = "apple";    // e.g. 0xff0
char *str1 = str + 1;   // e.g. 0xff1
char *str3 = str + 3;   // e.g. 0xff3

cout << str;            // apple
cout << str1;           // pple
cout << str3;           // le
```

DATA SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
// nums points to an int array
int *nums = …               // e.g. 0xff0
int *nums1 = nums + 1;  // e.g. 0xff4
int *nums3 = nums + 3;  // e.g. 0xffc

cout << *nums;              // 52
cout << *nums1;             // 23
cout << *nums3;             // 34
```

Memory stack

Address	Value
	…
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	…

Pointer Arithmetic

Pointer arithmetic does *not* work in bytes. Instead, it works in the *size of the type it points to*.

```
// nums points to an int array
int *nums = …               // e.g. 0xff0
int *nums3 = nums + 3;      // e.g. 0xffc
int *nums2 = nums3 - 1;     // e.g. 0xff8

cout << *nums;              // 52
cout << *nums2;             // 12
cout << *nums3;             // 34
```

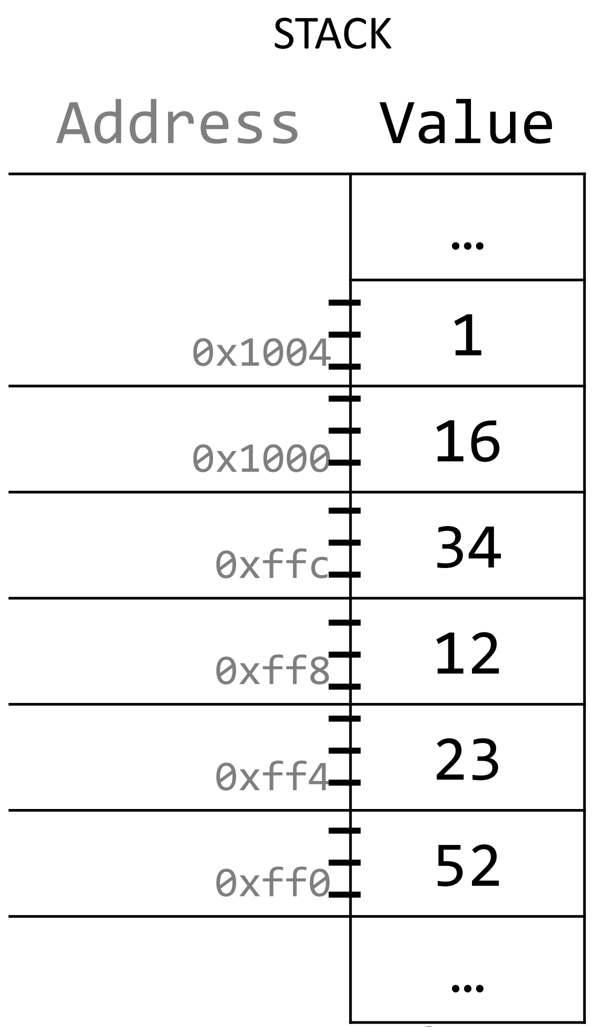
Memory stack

Address	Value
	…
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	…

Pointer Arithmetic

Hence, pointer arithmetic with two pointers does *not* give the byte difference. Instead, it gives the number of *places* they differ by.

```
// nums points to an int array
int *nums = …                // e.g. 0xff0
int *nums3 = nums + 3;       // e.g. 0xffc
int diff = nums3 - nums;     // 3
```



Pointer Arithmetic

When you use bracket notation with a pointer, you are actually *performing pointer arithmetic and dereferencing*:

```
char *str = "apple";    // e.g. 0xff0

// both of these add two places to str,
// and then dereference to get the char there.
// E.g. get memory at 0xff2.
char thirdLetter = str[2];        // 'p'
char thirdLetter = *(str + 2);   // 'p'
```

DATA SEGMENT

Address	Value
	…
0xff5	'\0'
0xff4	'e'
0xff3	'l'
0xff2	'p'
0xff1	'p'
0xff0	'a'
	…

Pointer Arithmetic

How does the code know how many bytes it should look at once it visits an address?

```
int x = 2;
int *xPtr = &x;      // e.g. 0xff0

// How does it know to print out just the 4 bytes at xPtr?
cout << *xPtr; // 2
```

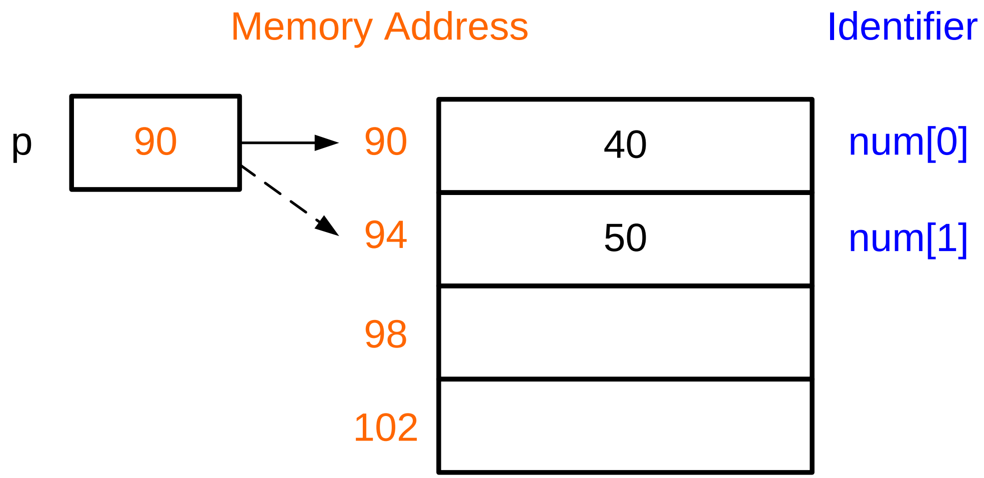
Pointer Arithmetic

- How does the code know how many bytes it should add when performing pointer arithmetic?

```
int nums[] = {1, 2, 3};

// How does it know to add 4 bytes here?
int *intPtr = nums + 1;

char str[7];
strcpy(str, "CS2310");

// How does it know to add 1 byte here?
char *charPtr = str + 1;
```

Pointer Arithmetic

- At compile time, the compiler can figure out the sizes of different data types, and the sizes of what they point to.
- For this reason, when the program runs, it knows the correct number of bytes to address or add/subtract for each data type.




```
int num[2] = {40,50};
int *p;
p = num;   /* p points to 90
*p = 400;
++p;          /*p points to 94 */
*p = 500;
```

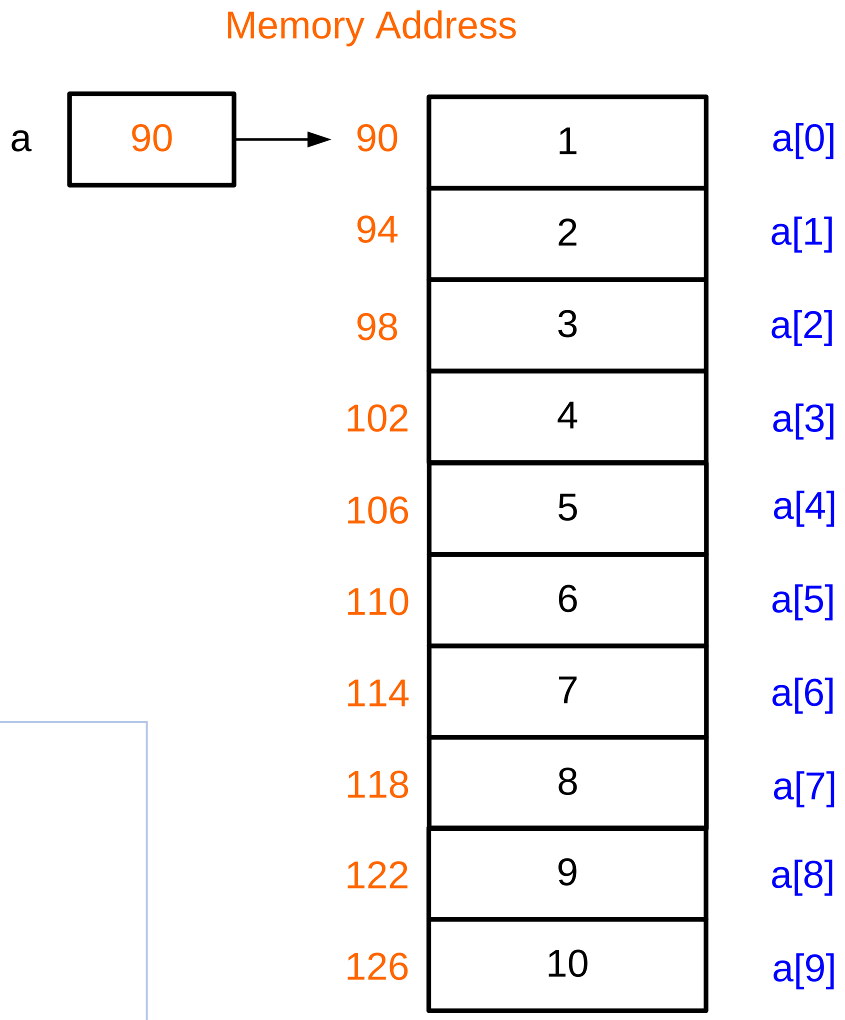
++p increments the content of p (an address) by **sizeof**(int) bytes

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 ith element of the array
*num	num[0]	The value of the 0th element of the array
*(num+i)	num[i]	The value of the ith element of the array
(*num)+i	num[0]+i	The value of the 0th element of the array plus i

Example: Pointer Arithmetic

```
#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);    //sum+=a[i];
    cout << sum;      // 55 is printed
    return 0;
}
```



`a+0` is the address of `a[0]`
`a+1` is the address of `a[1]`
...
`a+i` is the address of `a[i]`

So, `*(a+i)` means `a[i]`

Exercise: Pointer arithmetic

Suppose we use a variable `str` as follows:

```
// execute as below
A  str = str + 1;
B  str[1] = 'u';
C  cout << str;
```

For each of the following initializations:

- Will there be a compile error/runtime error?
- If no errors, what is printed?

1. ```
   char str[7];
   strcpy(str, "Hello1");
   ```
2. (Optional)
   ```
   char *str = "Hello2";
   ```
3. ```
   char arr[7];
   strcpy(arr, "Hello3");
   char *str = arr;
   ```
4. (Optional)
   ```
   char *ptr = "Hello4";
   char *str = ptr;
   ```

Exercise: Pointer arithmetic

Suppose we use a variable `str` as follows:

```
// execute as below
A  str = str + 1;
B  str[1] = 'u';
C  cout << str;
```

For each of the following initializations:

- Will there be a compile error/runtime error?
- If no errors, what is printed?

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

Line A: Compile error (cannot reassign array)

2.
```
char *str = "Hello2";
```

Line B: Runtime error (modify string literal on data segment)

3.
```
char arr[7];
strcpy(arr, "Hello3");
char *str = arr;
```

Prints eulo3

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

Line B: Runtime error (modify string literal on data segment)

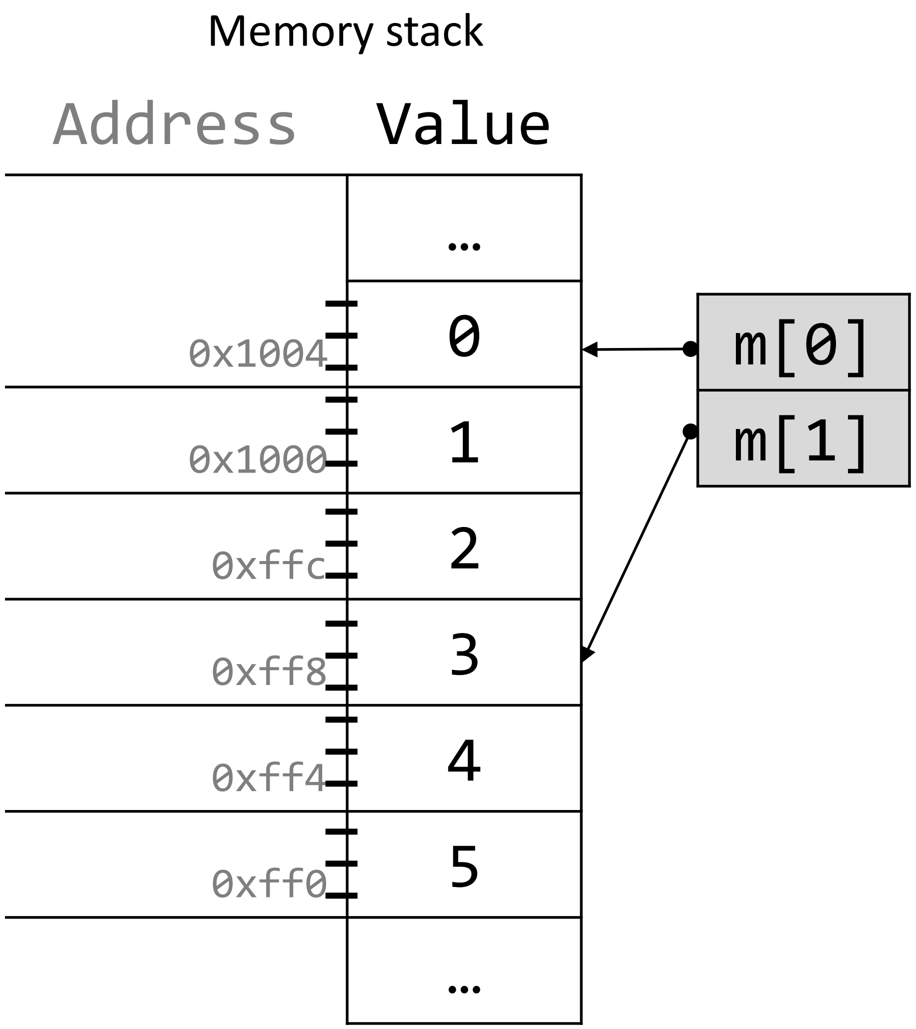
Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Double pointer & Pointer reference
- Const Pointer
- 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";
}
```



Pointer Array

Aside from integer, you can make an array of pointers to e.g. group multiple strings together:

```
char *stringArray[5];    // space to store 5 char *s
```

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

Pointer Array

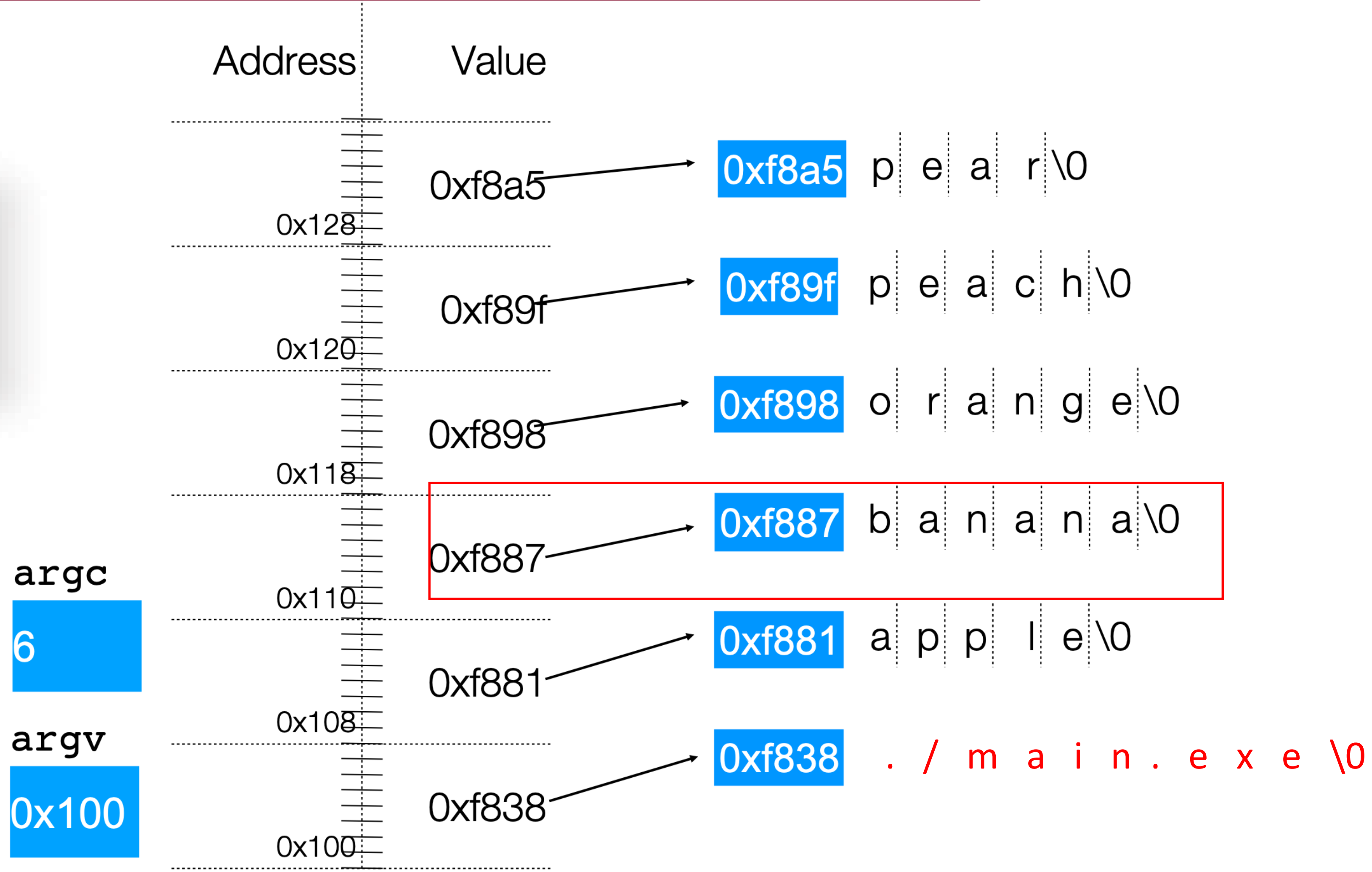
```
./main.exe apple banana orange peach pear
```

```
#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;
}
```

Pointer Array

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

What is the value of argv[2] in this diagram?

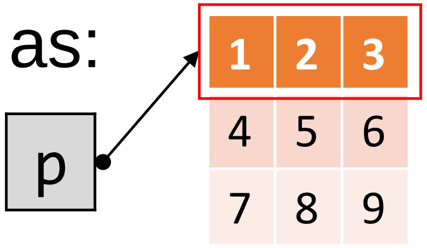


Array Pointer

- 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 pointer array (int* p[3])

```
int (*p)[3] = arr;  // a pointer to an entire 1D array of size 3, not just a single integer.
cout << *(*(p+1)+2) << endl; // p[1][2]
cout << *(p[2]+1)   << endl; // p[2][1]
```

Recap: 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
- Double pointer & Pointer reference
- Const pointer
- Dynamic memory allocation

Double 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;
```

Double 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;
```

Why Need Double Pointer?

- Example: write a program to skip leading spaces in a string
- Does the right-side program work? Why?

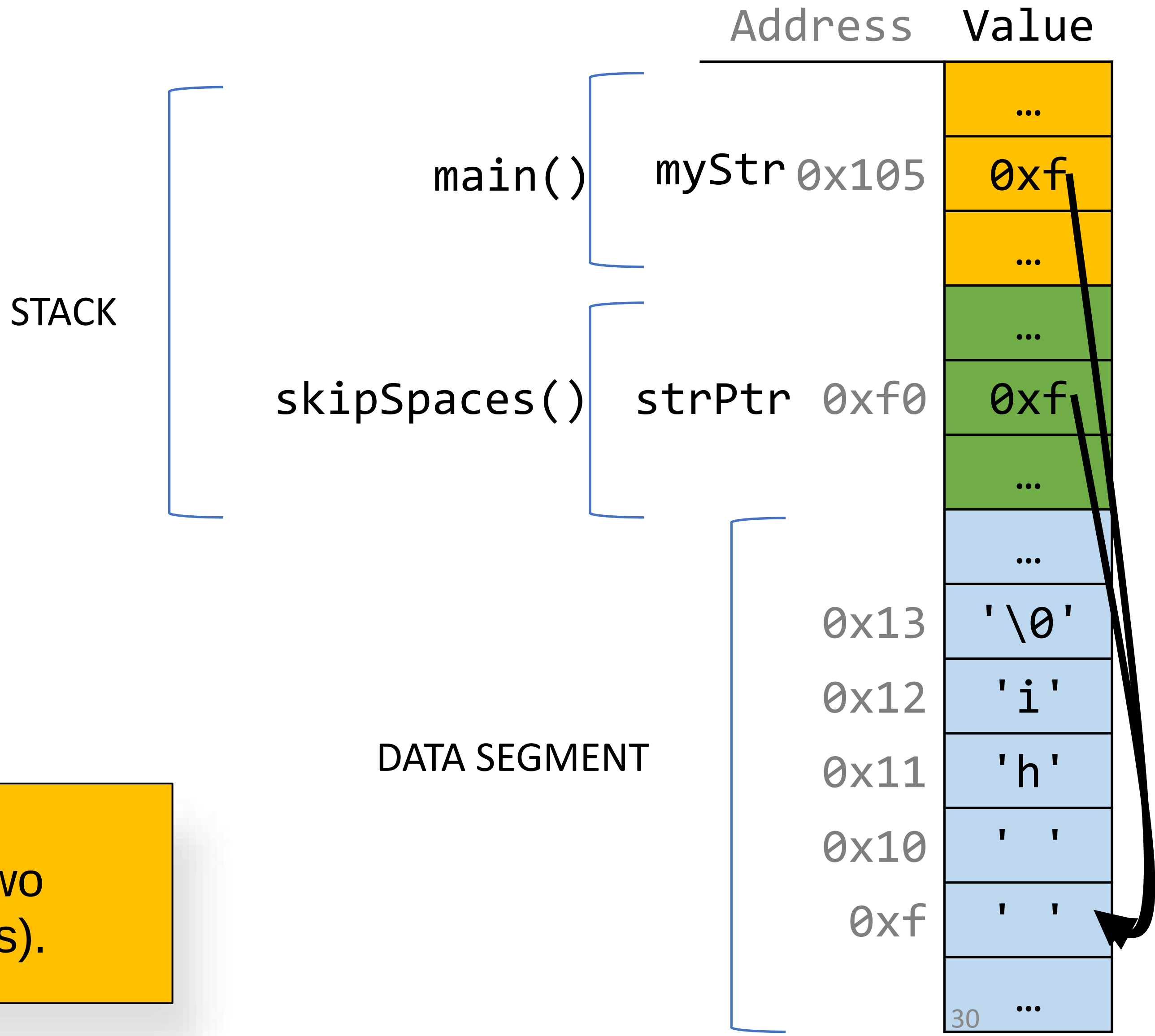
```
void skipSpaces(char *strPtr) {
      while (*strPtr == ' ')
            strPtr++;
      cout << strPtr << endl;
}

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

This advances skipSpace's own **copy of the string pointer**, not the instance in main.

Making Copies

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

int main() {
        char str[] = "  hi";
        char *myStr = str;
        skipSpaces(myStr);
        cout << myStr;        //  '  hi'
        return 0;
}
```

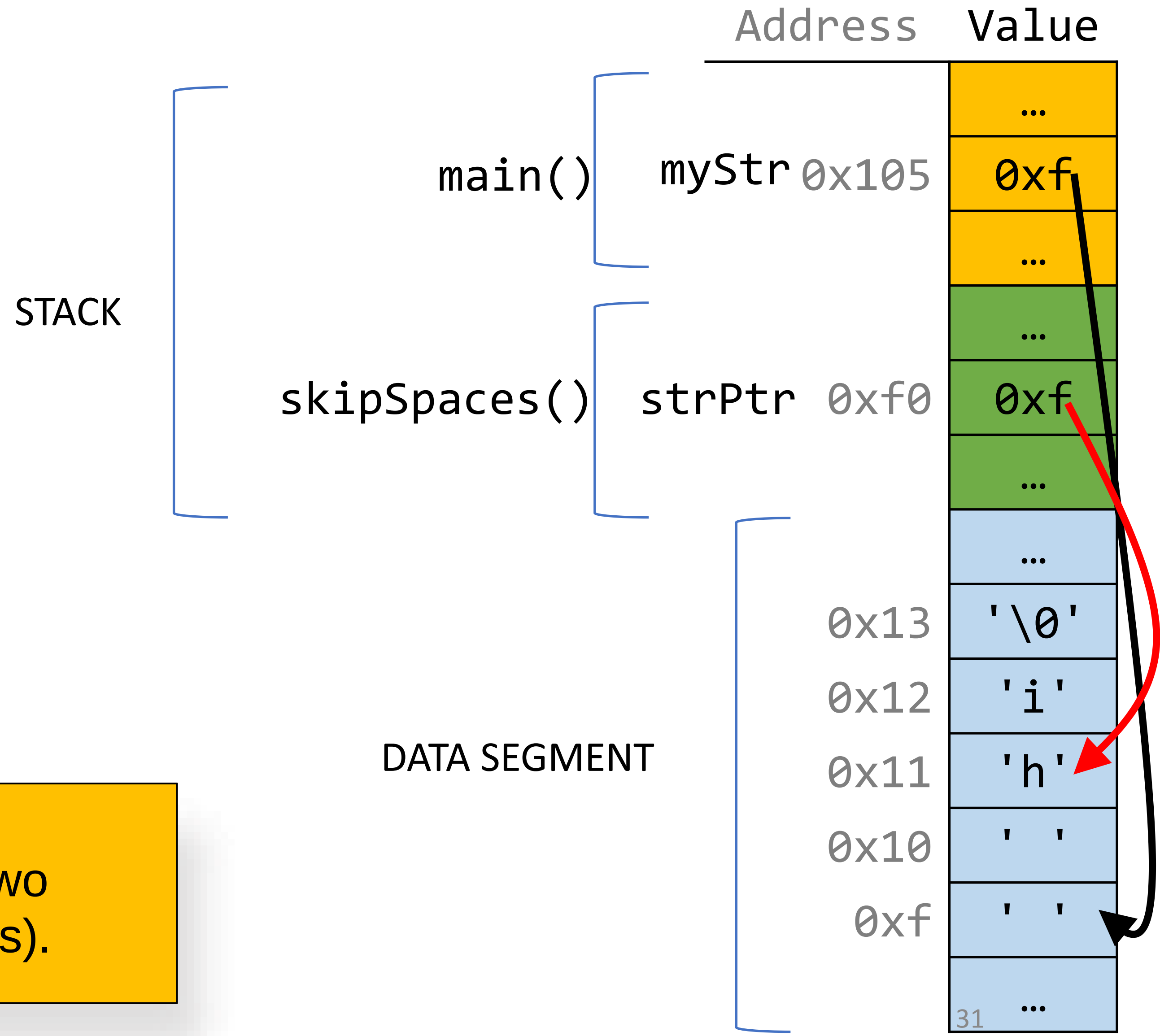
When you pass by pointer, C++ passes a copy of that pointer (two pointers with different addresses).



Making Copies

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

int main() {
      char str[] = "  hi";
      char *myStr = str;
      skipSpaces(myStr);
      cout << myStr;        //  '  hi'
      return 0;
}
```

When you pass by pointer, C++ passes a copy of that pointer (two pointers with different addresses).



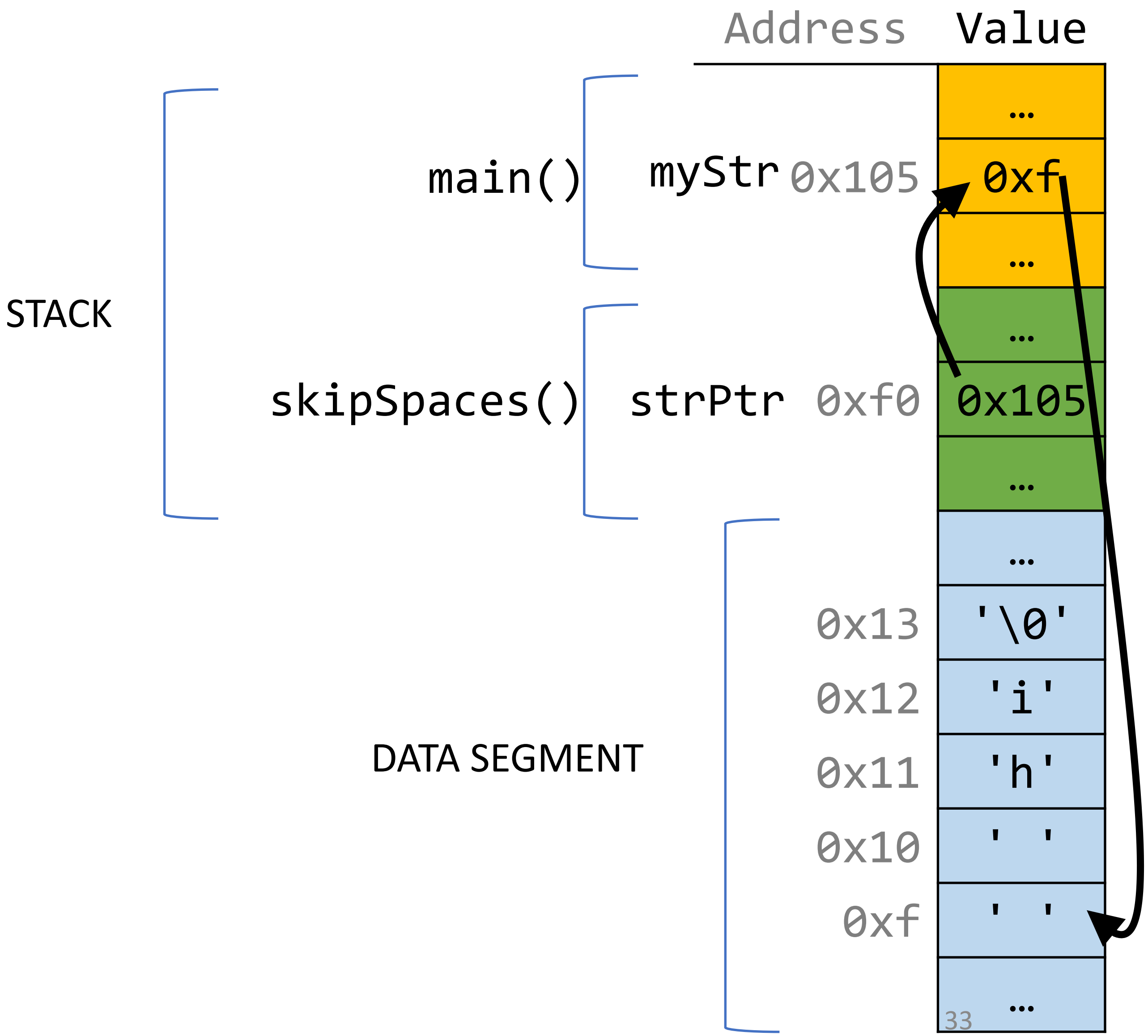
Why Need Double Pointer?

- Example: write a program to skip leading spaces in a string
- We want the called function to modify the pointer, so …

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

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

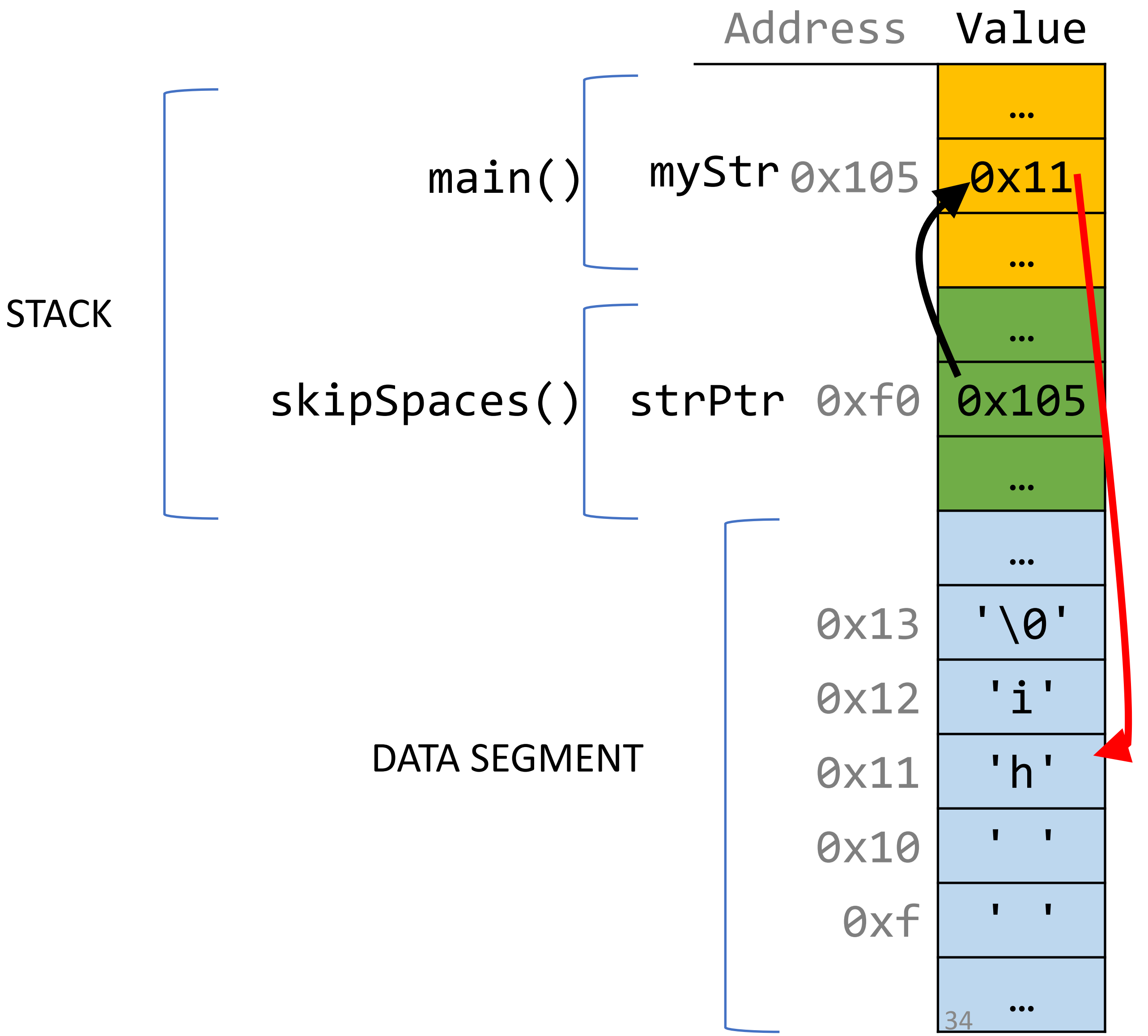
Pointers to Strings

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

int main() {
      char str[] = "  hi";
      char *myStr = str;
      skipSpaces(&myStr);
      cout << myStr;          //  'hi'
      return 0;
}
```



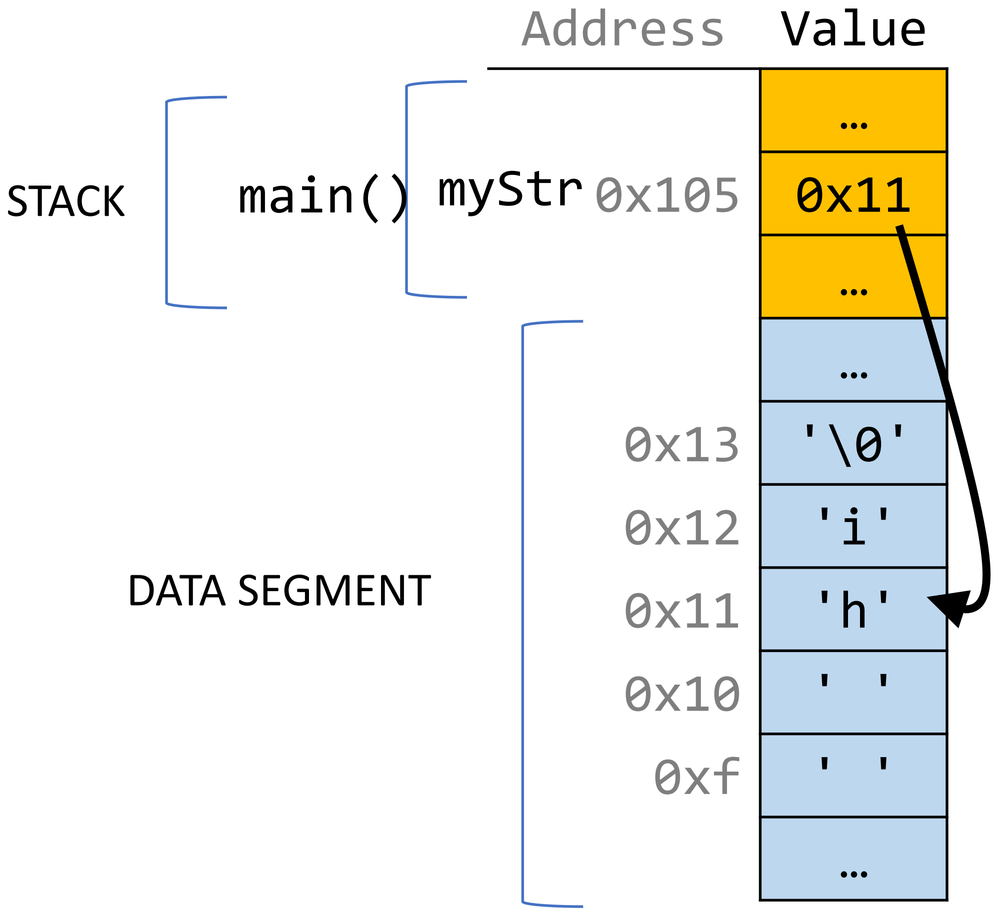
Pointers to Strings

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

int main() {
      char str[] = "  hi";
      char *myStr = str;
      skipSpaces(&myStr);
      cout << myStr;            //  'hi'
      return 0;
}
```



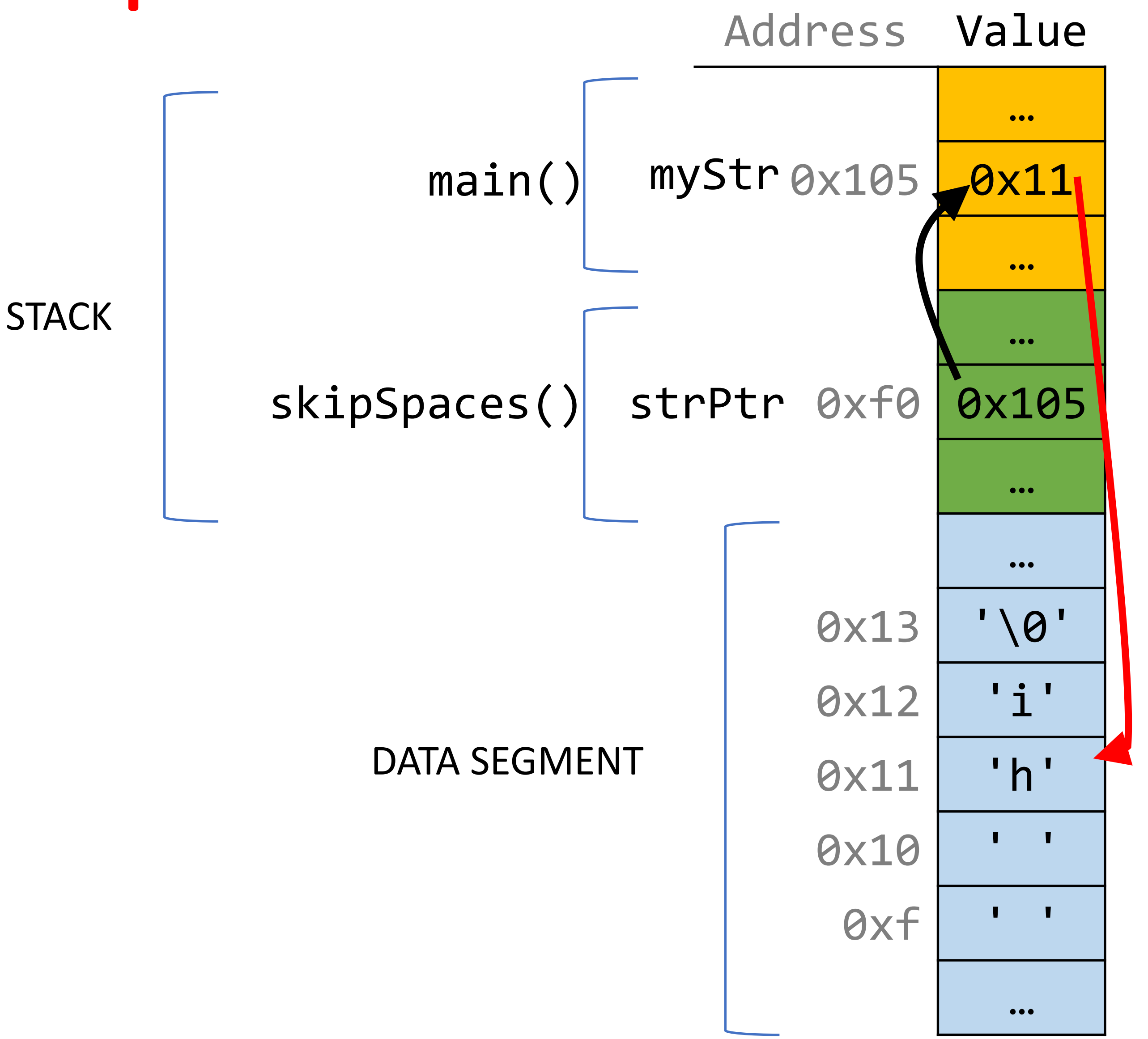
Pointers to Strings

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

int main() {
      char str[] = "  hi";
      char *myStr = str;
      skipSpaces(&myStr);
      cout << myStr;          //  'hi'
      return 0;
}
```



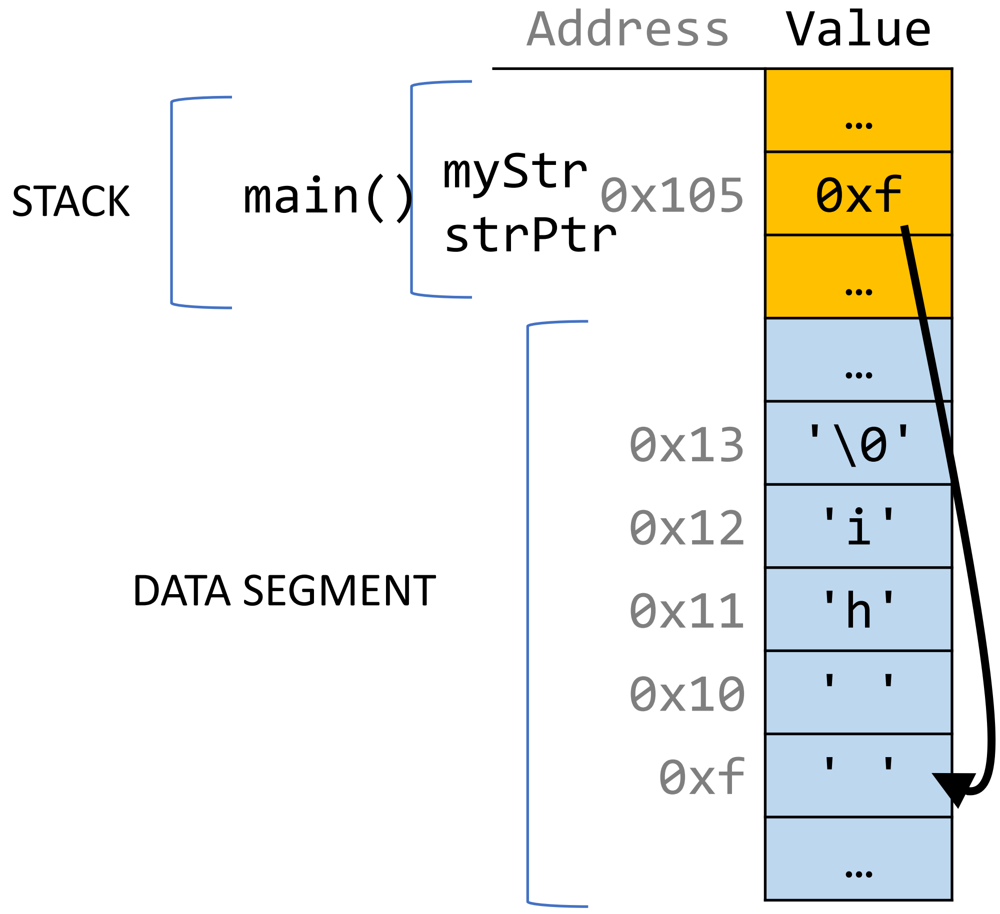
Exercise: What's the output?

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

int main() {
      char str[] = "  hi";
      char *myStr = str;
      skipSpaces(&myStr);
      cout << myStr << endl;
      cout << &myStr << endl;
      return 0;
}
//  char* will be treated as a cstring
```



Alternative: Pass by Pointer Reference

```
void skipSpaces(char* &strPtr) {
        while (*strPtr == ' ')
                strPtr++;
        cout << strPtr << endl; //  'hi'
}

int main() {
        char str[] = "  hi";
        char *myStr = str;
        skipSpaces(myStr);
        cout << myStr;          //  'hi'
        return 0;
}
```



Double Pointer vs Pointer Reference

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

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

Outline

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

Pointer to Const

Most often, we need to use **const** with pointers to indicate that the data that is pointed to cannot change.

```
char str[6];

strcpy(str, "Hello");
const char *s = str;

// Cannot use s to change characters it points to
s[0] = 'h';

const char *str2 = "Hello";     // Good practice. Why?
```

Pointer to Const

Sometimes we use **const** with pointer parameters to indicate that the **function** will not / should not change what it points to.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
     int count = 0;
     for (int i = 0; i < strlen(str); i++) {
          if (isupper(str[i])) {
               count++;
          }
     }
     return count;
}
```

Pointer to Const

By definition, C++ gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
     // compiler warning and error
     char *strToModify = str;
     strToModify[0] = …
}
```

Pointer to Const

By definition, C++ gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify. **Think of const as part of the variable type**.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
      const char *strToModify = str;
      strToModify[0] = …
}
```

Outline

- Pointer arithmetic
- Pointer array vs Array pointer
- Pointer of pointer & pointer reference
- Const Pointer
- 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 program 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:

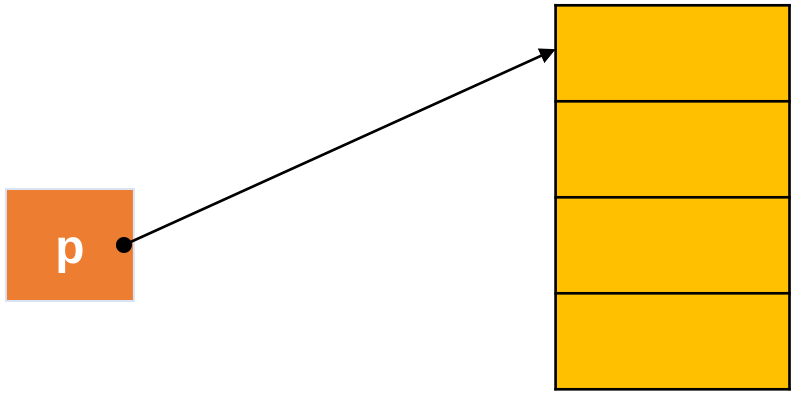
```
if (ptr)        // same as if (ptr != NULL)
if (!ptr)       // same as if (ptr == NULL)
```

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;
```

Dynamic Memory Walkthrough

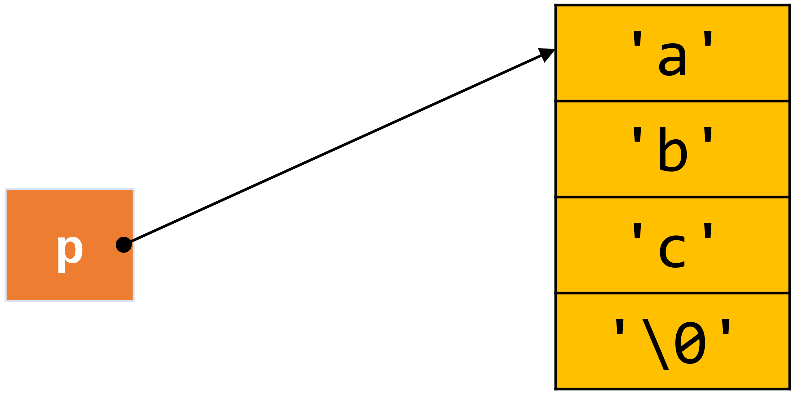
```
char *s1 = NULL;
s1 = new char[4];
cin >> s1; // input "abc"
cout << s1;
delete [] s1;
s1 = new char[6];
cin >> s1;
cout << s1;
delete [] s1;
s1 = NULL;
```



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

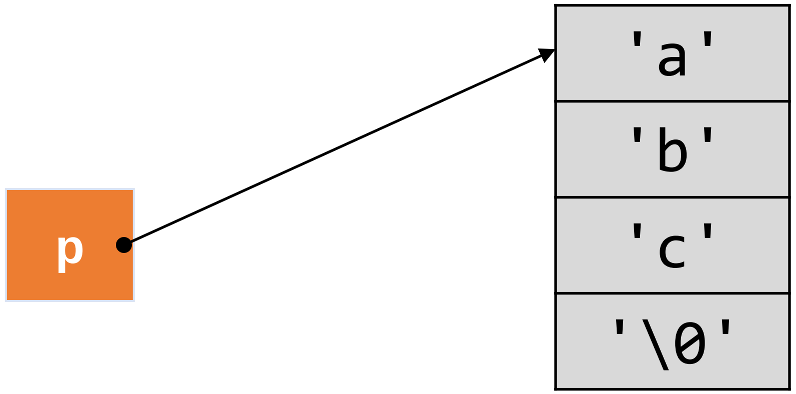
Dynamic Memory Walkthrough

```
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;
```

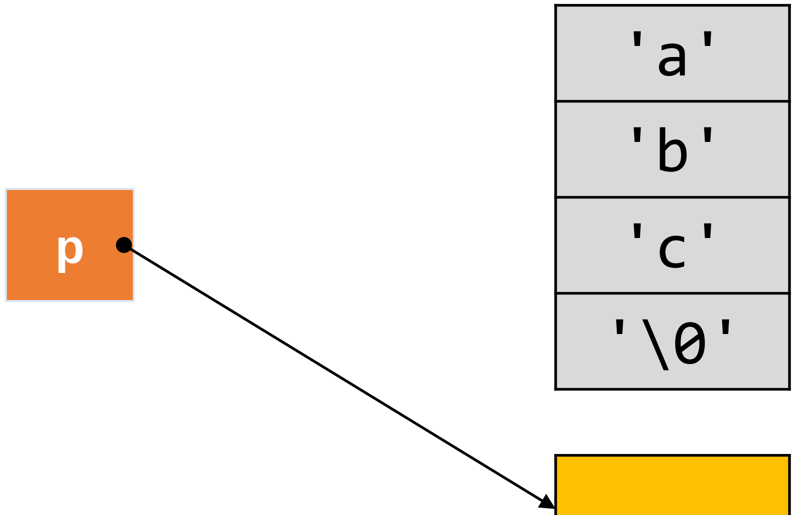


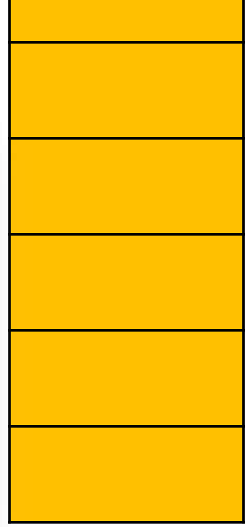
Grey memory means the block of memory is free and can be used to store other data.

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;
```

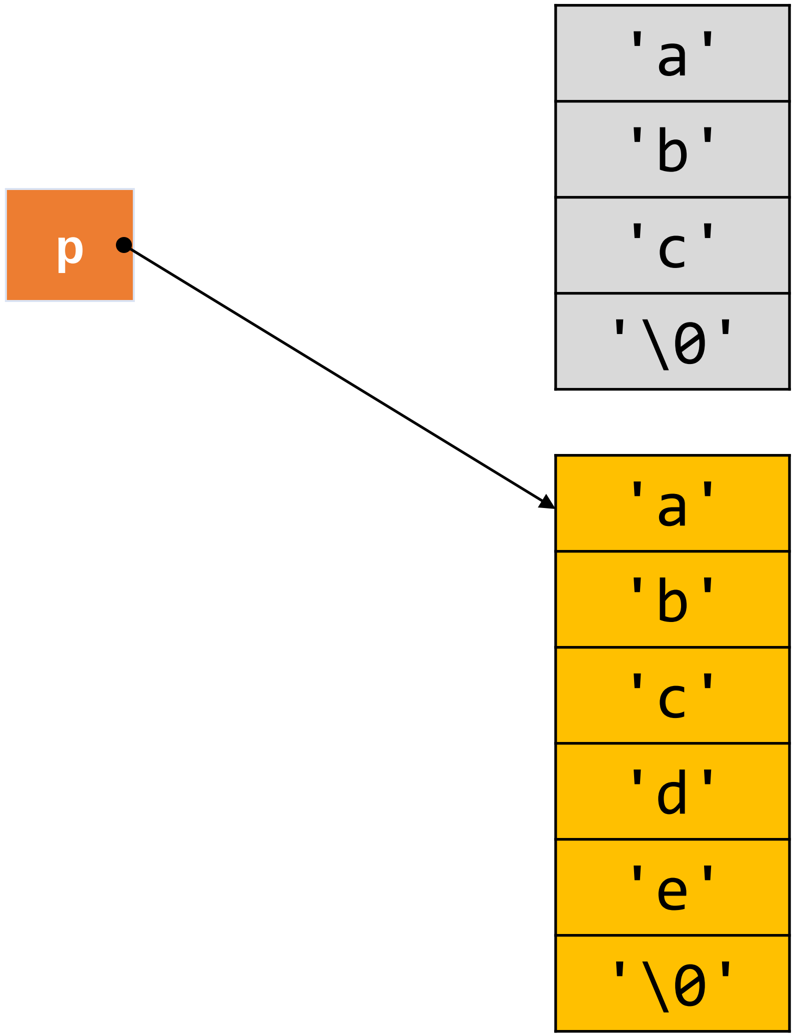




new dynamically allocates 6 bytes of memory. new returns a pointer to the 1st byte of the chunk of memory, which is assigned to s1

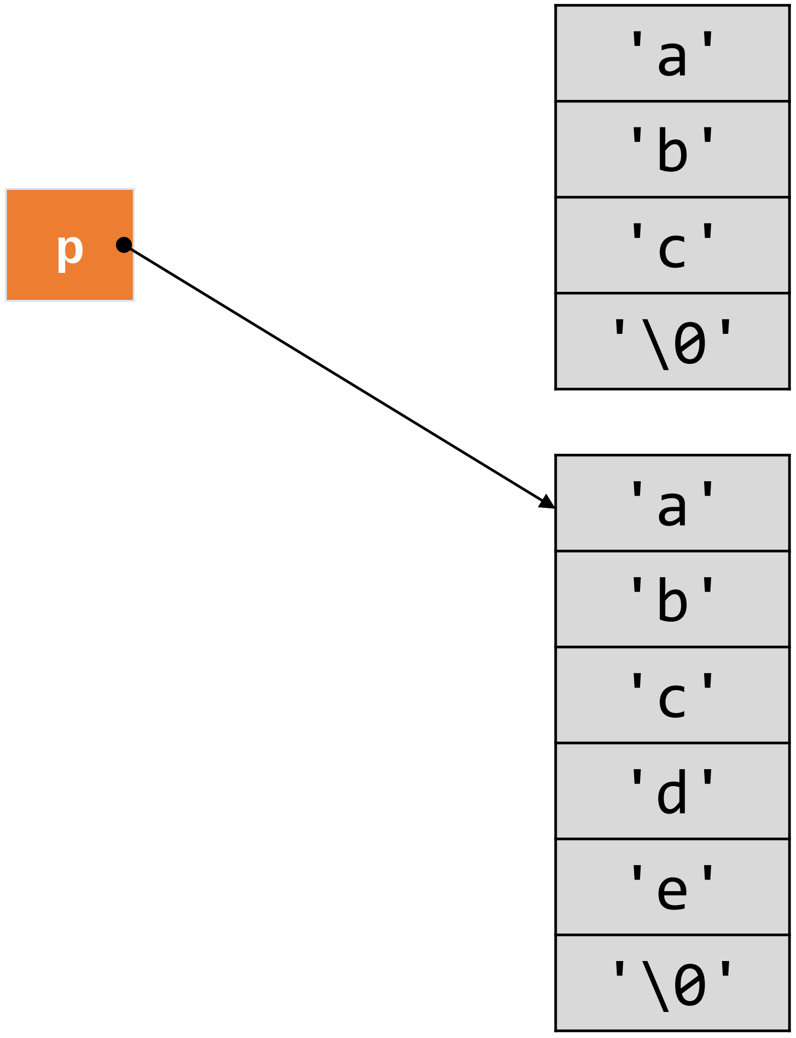
Dynamic Memory Walkthrough

```
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
```

p

'a'
'b'
'c'
'\0'

'a'
'b'
'c'
'd'
'e'
'\0'

Example

- score.txt contains the scores of 3 different courses for n students.
  - the first line of score.txt gives the value of n
  - reads all the scores, find all the students who have a failed score and output their scores for every course

- We can use dynamic memory allocation to solve the problem
  - As the number of the students is read from the input, we cannot define a normal 2D array (array size is not a constant).

score.txt :

```
4 3
85 89 64
93 82 94
55 92 59
59 88 70
```

```cpp
ifstream fin("score.txt");
if (fin.fail())
  exit(1);
int n, m;
fin >> n >> m;
int **p = new int*[n];
for (int i = 0; i < n; i++) {
  p[i] = new int[m];
  for (int j = 0; j < m; j++)
    fin >> p[i][j];
}
fin.close();
```

```cpp
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    if (p[i][j] < 60) {
      for (int k = 0; k < m; k++)
        cout << p[i][k] << ' ';
      cout << endl;
      break;
    }
  }
}
for (int i = 0; i < n; i++) {
  delete [] p[i];
}
delete[] p;
```

Summary

- Access of array via bracket notation actually performs *pointer arithmetic and dereferencing*
- Difference between Pointer Array and Array Pointer (2D Array)
- Double pointers in functions and Pass by Pointer Reference
- Const Pointer for read-only data, e.g., string literal
- Keywords and **Syntax** of Dynamic memory allocation