CS2311 Computer Programming

LT09: Pointer I

Computer Science, City University of Hong Kong

Semester B 2022-23

Review: string

- C string basics
- Reading and printing C strings
- Common string functions
- Safety of string functions

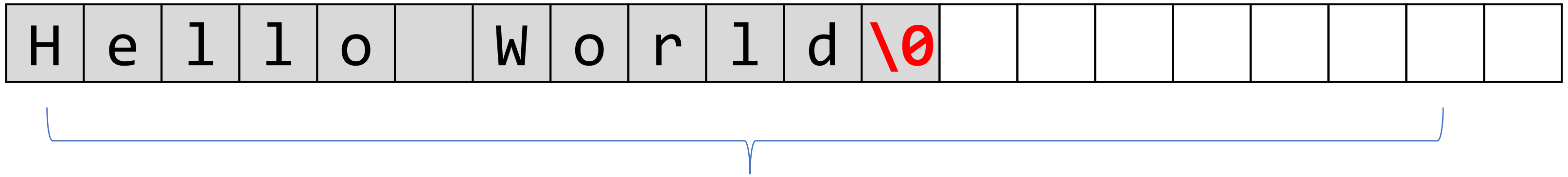
Review: cstring vs std::string

- In C++, there are two types of strings
  - cstring: inherited from the C language
    - #include <cstring>
  - string: *class* defined in std library
    - #include <string>
    - Class and object (introduced in later lecture)

Review: C String

- A C string is a char array terminated by '\0'
- '\0': null character representing the end-of-string sentinel
- Consider the definition and initialization of char str[20]

```c
char str[20] = "Hello World"; // '\0' will be added automatically
```



H	e	l	l	o		W	o	r	l	d	\0								

str may store a string with maximum of 19 characters

Review: C String Declaration and Initialization

- Declare a C string with one more character than needed
    - reserve one slot for '\0'
- A string can be declared in two ways
    - With initialization: char identifier[] = string constant / string literal;

```
e.g., char studentID[] = "a1234567";
      char HKID[] = "a123456(7)";
```

- Without initialization: char identifier[required_size+1];

```
e.g., char studentID[8+1];
      char HKID[10+1];
```

Review: Reading C Strings

- `cin >> str` will terminate when a whitespace character is encountered
  - whitespace: space, tab, newline …

- Suppose “Hello world” is the input

```
char s1[20], s2[10];
cin  >> s1;   // user input "hello world\n"
              // cin reads "hello" and stops when ' ' is encountered;
              // s1 gets "hello", '\0' is automatically added
              // "world\n" is stored in buffer to be consumed later
cin  >> s2;   // since there's content left in buffer, cin will read buffer first
              // i.e., no user input is needed
              // cin reads "world" in buffer and stops when '\n' is encountered
              // s2 gets "world", '\0' is automatically added
cout << s1;   // will print "hello"
cout << s2;   // will print "world"
```

Reading a Line: get() Loop

- **cin >> str** stops when a whitespace is encountered
  - How to get a line of chars from user input (before '\n' is encountered)?
- **get():** member function of cin to read in one character from input
  - **>>** skipping over whitespace but **get()** won't
- syntax: char c;
  cin.get(c);

```
#include <iostream>
using namespace std;

// read user input to str, until
// the end of line (i.e., '\n') is reached
// or str is full

int main() {
  char str[20];
  int i = 0;
  char c;
  do {
     cin.get(c);
     cout << c;
     str[i++] = c;
  } while (c!='\n' && i<20);
  return 0;
}
```

Review: Reading a Line: getline

- **getline():** predefined member function of cin to read a line of text (including space)

- Two arguments:
  - a C string variable to receive the input
  - size of the C string

```
#include <iostream>
using namespace std;
int main() {
  char s[20];
  while (true) {
    cin.getline(s, 20);
    cout << "\"" << s << "\"" << "\n";
  }
  return 0;
}
```

Review: strlen

- **strlen(str)**: returns the number of chars (before '\0') in C string **str**
  - '\0' does NOT count towards the length
- In comparison, recall that sizeof returns array size (number of bytes)

```
char myStr[20] = "Hello World!";
int len = strlen(myStr);
int siz = sizeof(myStr);
cout << len << "\n"; // 12
cout << siz << "\n"; // 20
```

Review: strcpy

- **strcpy(dst, src)**: copies the characters of string **src** into string **dst**, stops when '\0' is encountered in **src**

```
char s1[6];
strcpy(s1, "hello");
char s2[6];
strcpy(s2, s1);
s2[0] = 'c';
cout << s1 << endl; // hello
cout << s2 << endl; // cello
```

Review: strcat

- We **cannot** concatenate C strings using +: this adds addresses!
- Instead, use strcat
  - **strcat(dst, src)** concatenates the contents of **src** into **dst,** i.e., copies the characters in **src** to the end of **dst**, until '\0' is encountered in **src**

```
char str1[13];
strcpy(str1, "hello ");
strcat(str1, "world!"); // removes old '\0', adds new '0' at the end
cout << str1;
```

Review: strcat

```
char str1[13];
strcpy(str1, "hello ");
char str2[7];
strcpy(str2, "world!");
strcat(str1, str2);
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
str1	'h'	'e'	'l'	'l'	'o'	' '	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

	0	1	2	3	4	5	6
str2	'w'	'o'	'r'	'l'	'd'	'!'	'\0'

Review: strcmp

strcmp(str1, str2) compare **str1** and **str2,** until

  - encounters a pair of characters that don't match
  - reaches the end of str1 or str2 (i.e., encounters '\0' in str1 or str2)

- Let **c1** and **c2** be the pair of characters in **str1** and **str2** that don't match
  - **< 0**: if **c1 < c2** (i.e., **str1** is smaller than **str2** in alphabet)
  - **> 0**: if **c1 > c2** (i.e., **str1** is greater than **str2** in alphabet)
  - **return 0** if **str1** and **str2** are identical

- e.g.,

```
cout << strcmp("abc",  "abc") << "\n"; //  0
cout << strcmp("abc", "abcd") << "\n"; // -1
cout << strcmp("abcd", "abc") << "\n"; //  1
cout << strcmp("abc",  "abd") << "\n"; // -1
```

Review: File I/O vs. Console I/O

- "Console I/O" is volatile,refers to "keyboard input/screen output"
- Files I/O is non-volatile
  - input file can be used again and again
  - output file retains results
- Allow off-line processing
- Useful for debugging especially when volume of data is huge

Review: File Streams

- File stream class in C++
  - #include <fstream> // similar with "#include <iostream>"
  - ifstream: stream class for file input, similar with cin
  - ofstream: stream class for file output, similar with cout

- To declare an objects of class ifstream or ofstream, use
  - ifstream fin;
  - ifstream infileName;
  - ofstream fout;
  - ofstream outFileName;

int variableA;

Review: ifstream

- To declare an ifsteam type/object
  - ifstream fin;
- To open a file for reading
  - fin.open("infile.dat"); // infile.dat is the filename
- To read the file content
  - fin >> x; // x is a variable
- To close the file
  - fin.close();

Review: Example of using File I/O

```
#include <fstream>
using namespace std;
int main() {
        ifstream finName;
        ofstream foutName;
        int x, y, z;
        finName.open("input.txt");
        foutName.open("output.txt");
        finName >> x >> y >> z;
        foutName << "The sum is " << x+y+z;
        finName.close();
        foutName.close();
        return 0;
}
```

input.txt

```
999 100 8
```

Review: fail()

```
fstream fin("test.txt");
if (fin.fail()) {
      cout << "fail to open "test.txt\n";
      exit(1);
}
```

// when an I/O operation fails, one may call exit() to abort the program execution

// the argument in exit() is returned to the calling party -- usually the OS

// typically, exit(1) is used to abort program when there's an error

Review: eof()

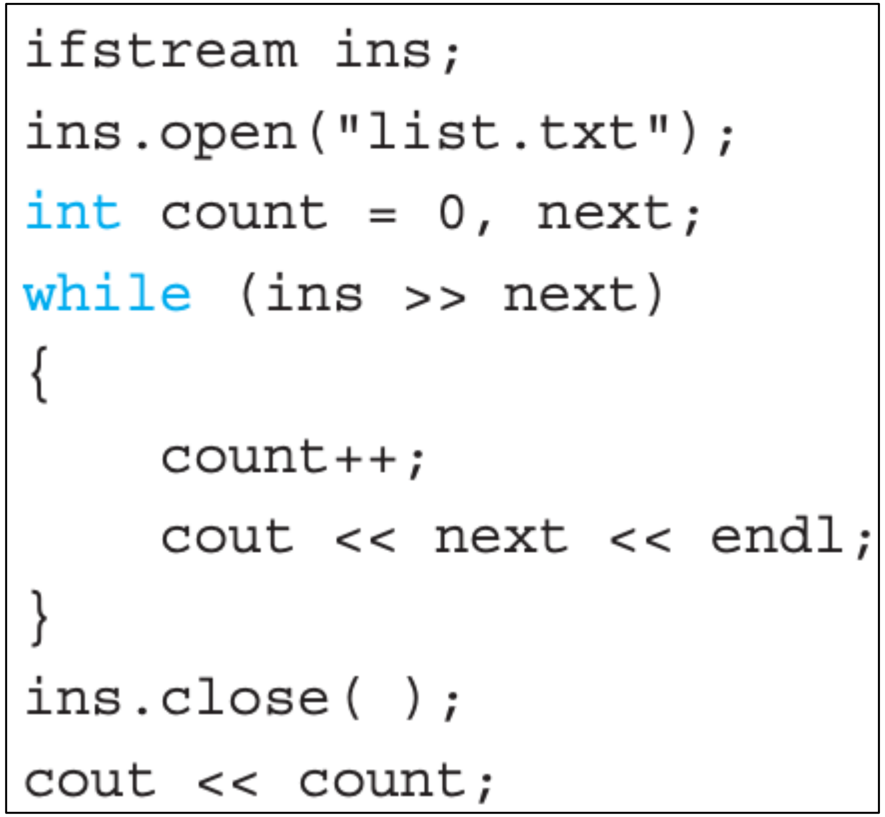
```
// dump the content from input.txt to output.txt
// assuming input.txt contains only integers
ifstream fin;
ofstream fout;
fin.open("input.txt");
fout.open("output.txt");
int x;
while (!fin.eof()) {
    fin >> x;
    fout << x << " ";
}
```

Review: Some hints:

```cpp
#include <iostream>
#include <cstring>
using namespace std;
int main() {
   char s[20];
   cin.getline(s, 20);
   for (int i = 0; s[i] != '\0'; i++) {
      if (s[i] >= 'A' && s[i] <= 'Z')      // uppercase letter
         cout << char('a' + s[i]-'A');     // convert to lowercase
      else if (s[i] >= 'a' && s[i] <= 'z') // lowercase letter
         cout << char('A' + s[i]-'a');     // convert to uppercase
      else                                 // other letters
         cout << s[i];
   return 0;
}
```

Exercise 1

- What output will be produced when the following lines are executed, assuming the file list.txt contains the data shown (and assuming the lines are embedded in a complete and correct program with the proper include and using directives)?



```
ifstream ins;
ins.open("list.txt");
int count = 0, next;
while (ins >> next)
{
     count++;
     cout << next << endl;
}
ins.close( );
cout << count;
```

```
1  2
        3
a
```

list.txt

1
2
3
3

Exercise 2

- What output will be produced when the following lines are executed, assuming the file contains the data shown (and assuming the lines are embedded in a complete and correct program with the proper include and using directives)?

```
a b c
```

input.txt

```
ifstream fin("input.txt");
int count = 0;
char next;
while (!fin.eof()) {
  fin >> next;
  cout << next << endl;
  count++;
}
fin.close();
cout << count << endl;
```

```
a
b
c
c
4
```

Exercise 3

- Write a C++ program to find the longest word in a given string (assume size is smaller than 3000)
- Example:
  Sample Input: C++ is a high level programming language.
  Sample Output: programming

Exercise 3

```
#include <iostream>
#include <cstring>
#include <fstream>
using namespace std;

int main() {
    char astring[3000];
    cin.getline(astring, 3000);
    int i, start = 0, longest = 0, longest_pos = 0;
    for (i = 0; astring[i] != '\0'; i++) {
        if (astring[i] == ' ') {
            start = i + 1;
        } else {
            if (i - start > longest) {
                longest = i - start;
                longest_pos = start;
            }
        }
    }
    for (i = longest_pos; i <= longest_pos + longest; i++)
      cout << astring[i];
    return 0;
}
```

Outlines

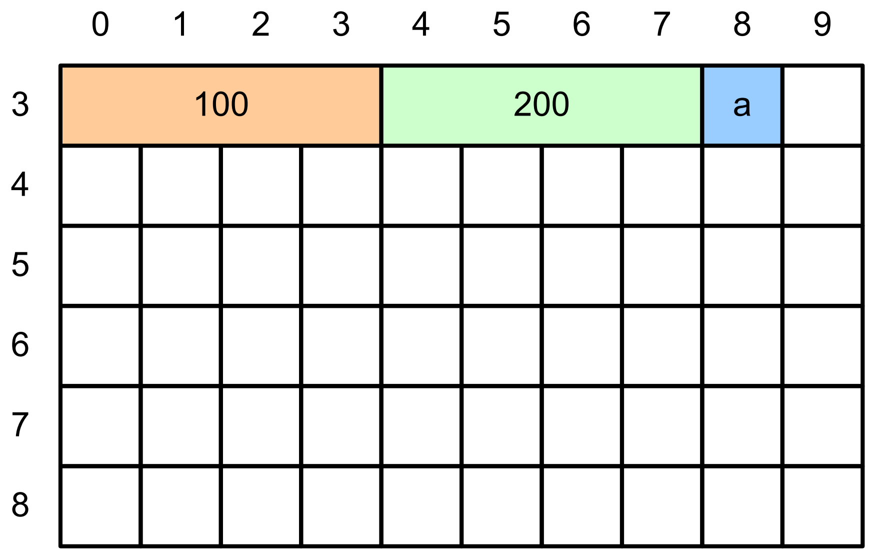
- Recap: variable and memory
- Pointer and its operations
- Pass by pointer
- Array and pointer

Recap: Variable and Memory

- Variable is used to store data that will be accessed by a program
- Normally, variables are stored in the **main memory**
- A variable has five attributes:
  - Value - the content of the variable
  - Type – data type, e.g., int, float, bool
  - Name - the identifier of the variable
  - **Address** - the memory location of the variable
  - Scope - the accessibility of the variable

Recap: Variable and Memory

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



	0	1	2	3	4	5	6	7	8	9
3	100				200				a	
4										
5										
6										
7										
8										

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

Recap: 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

A cstring "apple"

Address	Value
	…
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	…

Outlines

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

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

Why Study Pointer?

- C/C++ allows programmers to talk directly to memory
  - Highly efficient in early days
    - Because there is no **pass-by-reference** in C like in C++, pointers let us pass the memory address of data, instead of copying values
  - Other languages (like Java) manage memory automatically
    - runtime overhead, less efficient than human programmer
  - However, many higher-level languages today attain acceptable performance
  - Despite that, low-level system code still needs low-level access via pointers
    - hence continued popularity of C/C++

Definition of Pointer

- 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

- Pointer, like normal variable, has a type. The pointer type is determined by the type of the variable it points to

Basic Pointer Operators: & and *

```
int x = 2;

// Make a pointer that stores the address of x
// To declare an int pointer, place a "*" before identifier
// assign address of x to pointer (& is address operator here)

int *xPtr = &x;

// Dereference the pointer to get the value stored in that address
// (* is the dereference operator in this context)
cout << *xPtr;          // prints 2
```

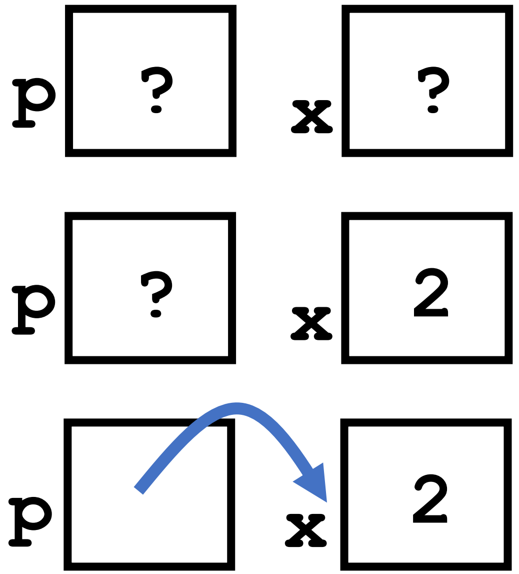
Basic Pointer Operators: & and *

& 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.*

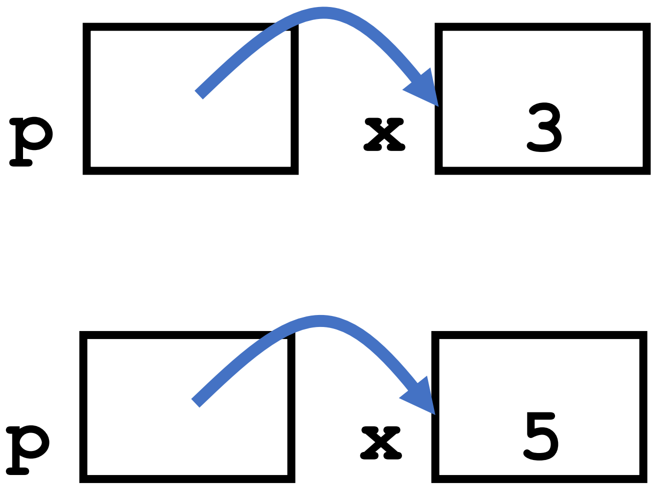
```
int *p,x;

x = 2;

p = &x;


cout << "p points to ";
cout << *p;
```



Basic Pointer Operators: & and *

- To write a value into memory using dereference operator *, we can use the dereference operator * on the left of assignment operator **=**

```
int x = 3;

int *p = x;

*p = 5;
```



Example

```
int x,y;              // x and y are integer variables
int main() {
     int *p1, *p2;   // p1 and p2 are pointers of integer typed
     x = 10; y = 12;
     p1 = &x;        // p1 stores the address of variable x
     p2 = &y;        // p2 stores the address of variable y
     *p1 = 5;        // p1 value unchanged but x is updated to 5
     *p2 = *p1+10;   // what are the values of p2 and y?
     return 0;
}
```

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

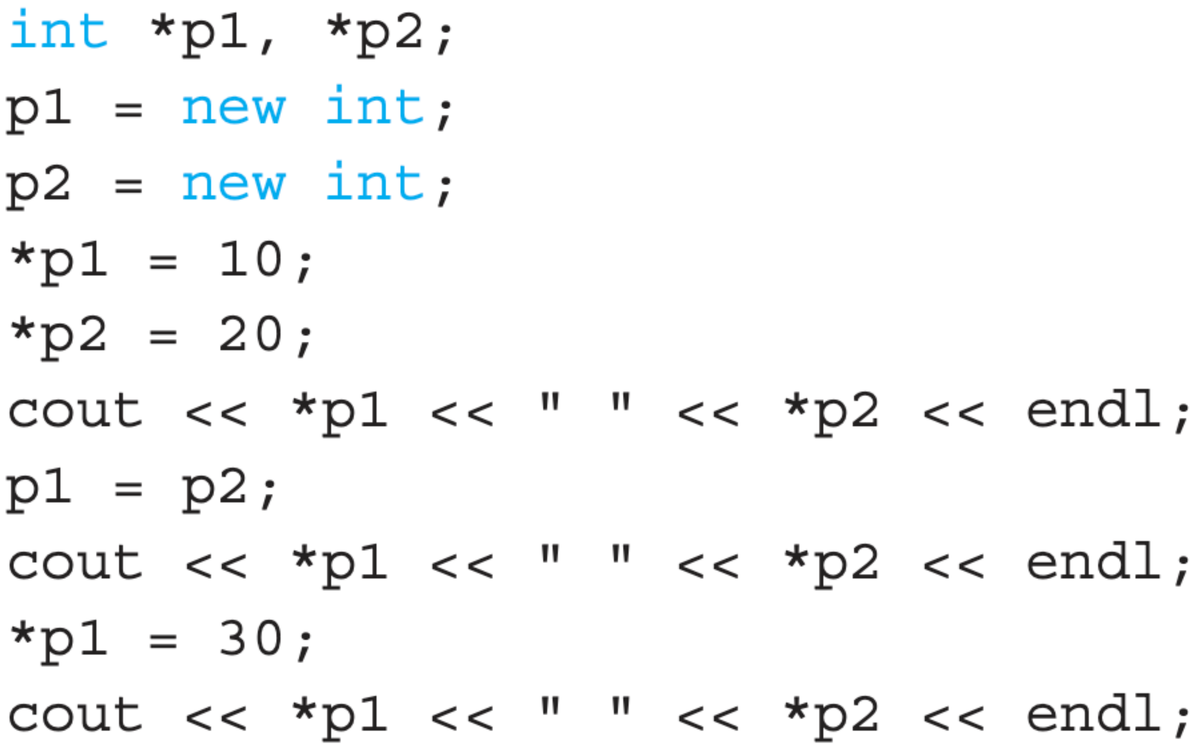
  int x = **p2*;

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;
```

Exercise

What is the output produced by the following code?



```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

Exercise

What is the output produced by the following code?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

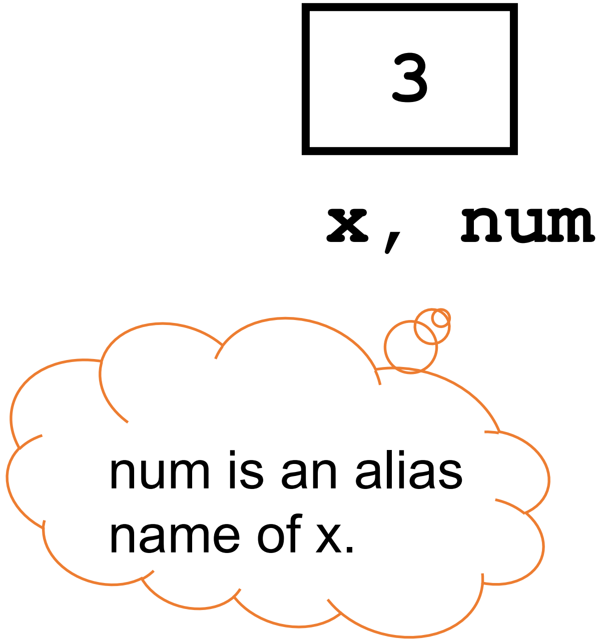
Outlines

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

Recap: Pass-by-Reference

& sign is called reference declarator in this context.

```
void myFunc(int& num) {
     num = 3;
}

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



num is an alias name of x.

Pass-by-Reference vs Pass-by-Pointer

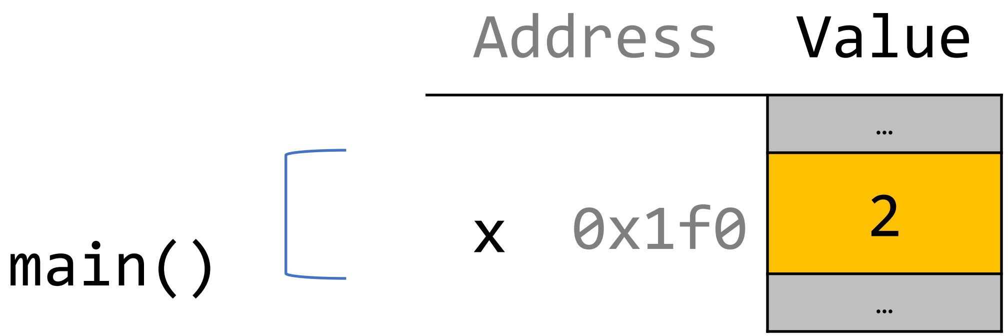
```
void myFunc(int& num) {
    num = 3;
}

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

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

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

Pass-by-Pointer

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

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

Memory stack



Address Value

main()

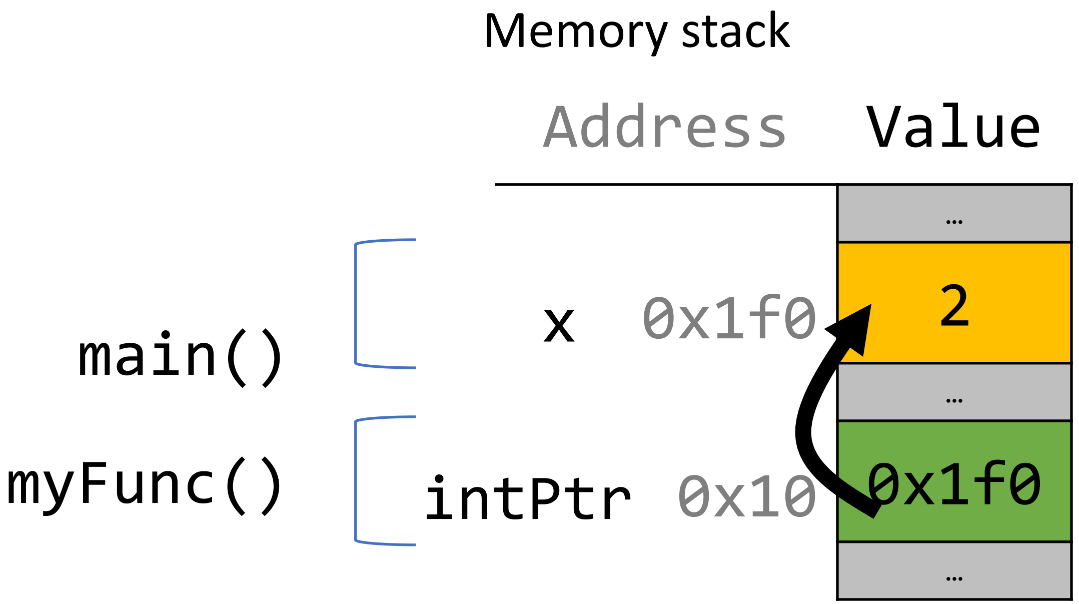
…

x 0x1f0 2

…

Pass-by-Pointer

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

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

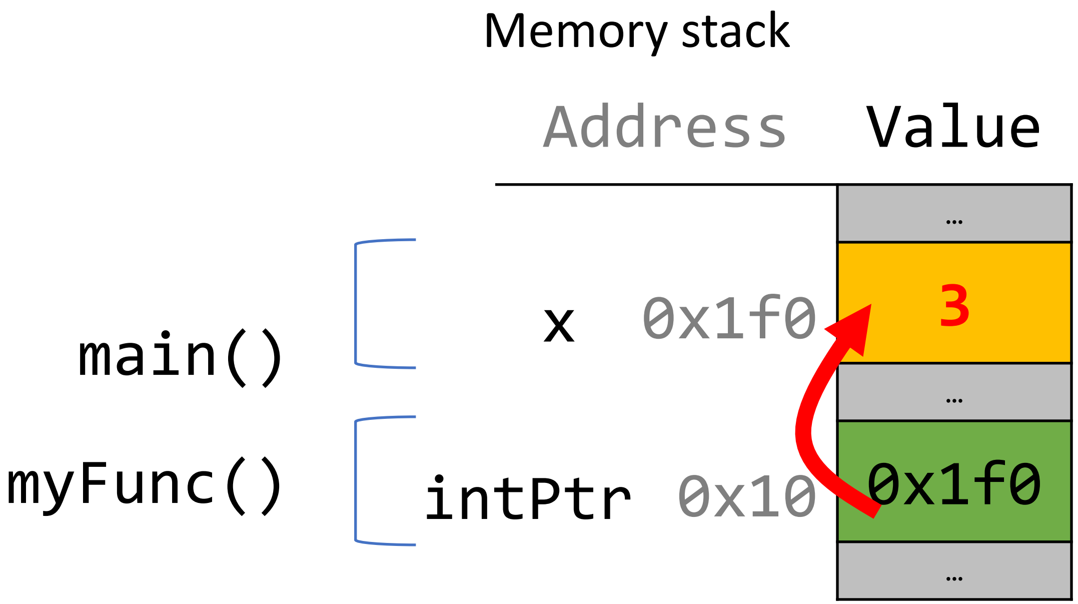


Memory stack

		Address	Value
			…
main()	x	0x1f0	2
			…
myFunc()	intPtr	0x10	0x1f0
			…

Pass-by-Pointer

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

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



Memory stack

		Address	Value
			…
main()	x	0x1f0	3
			…
myFunc()	intPtr	0x10	0x1f0
			…

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

Pass-by-Pointer

```
void doSth(char *a) {
    *a = 'a';
    *(++a) = 'b';
}
int main() {
    char str[] = "Hello";
    doSth(&str[1]);
    cout << str;
    return 0;
}
```

Can you tell me what the output will be?

Hablo

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**
- *How about pass-by-reference?*

Pass-by-Pointer vs Pass-by-Reference

```
void doSth(char *a) {
     *a = 'a';
     *(++a) = 'b';
}
int main() {
     char str[] = "Hello";
     doSth(&str[1]);
     cout << str;
     return 0;
}
```

```
void doSth(char &a) {
     a = 'a';
     ++a = 'b';
}
int main() {
     char str[] = "Hello";
     doSth(str[1]);
     cout << str;
     return 0;
}
```

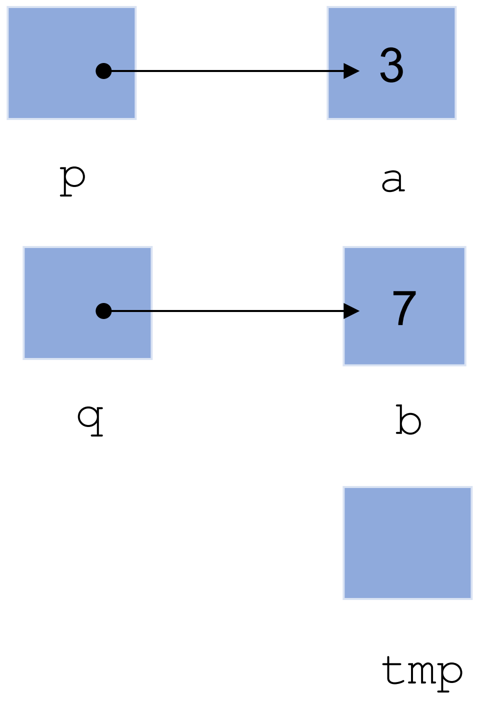
Pass-by-Pointer vs Pass-by-Reference

```
void doSth(char *a) {
     *a = 'a';
     *(++a) = 'b';
}
int main() {
     char str[] = "Hello";
     doSth(&str[1]);
     cout << str;
     return 0;
}
```

```
void doSth(char &a) {
     a = 'a';
     char *p = &a;
     *(++p) = 'b';
}
int main() {
     char str[] = "Hello";
     doSth(str[1]);
     cout << str;
     return 0;
}
```

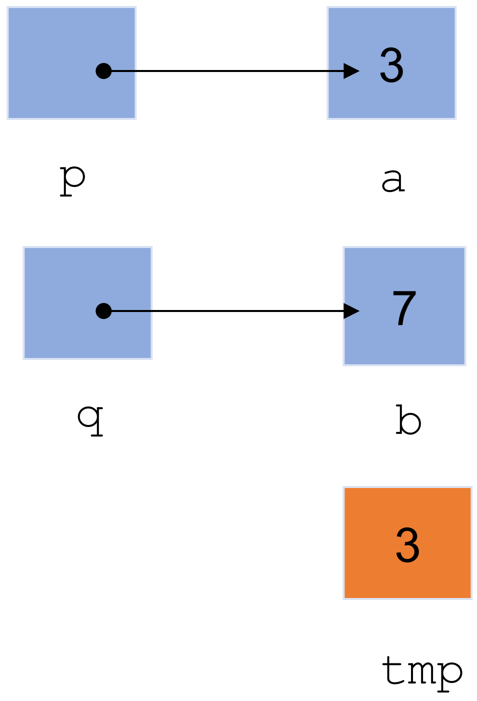
Example: Swapping Value

```
#include <iostream>
using namespace std;

void swap(int *p, int *q) {
   int tmp;
   tmp = *p;           /* tmp = 3 */
   *p = *q;            /* *p = 7 */
   *q = tmp;           /* *q = 3 */
}

int main() {
   int a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl;
    /* 7  3 is printed */
   return 0;
}
```



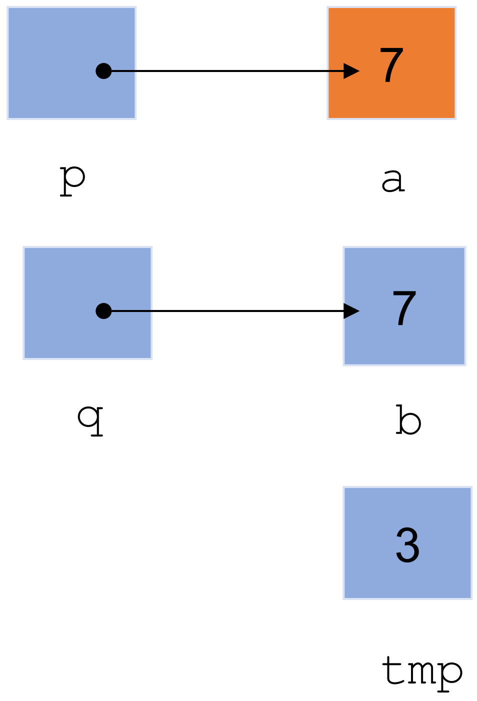
Example: Swapping Value

```
#include <iostream>
using namespace std;

void swap(int *p, int *q) {
   int tmp;
   tmp = *p;           /* tmp = 3 */
   *p = *q;            /* *p = 7 */
   *q = tmp;           /* *q = 3 */
}

int main() {
   int a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl;
    /* 7  3 is printed */
   return 0;
}
```



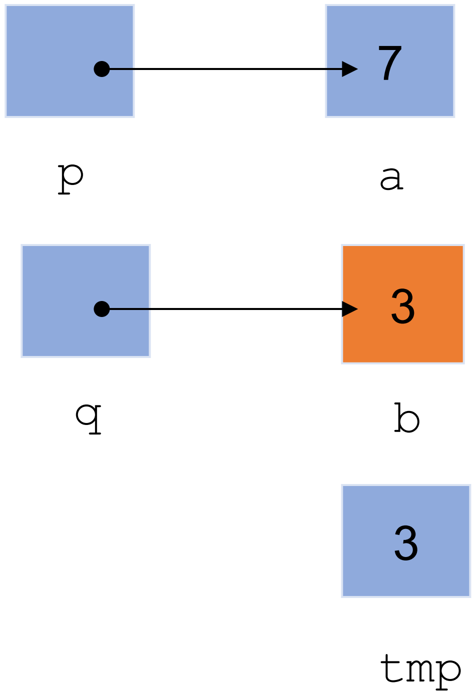
Example: Swapping Value

```
#include <iostream>
using namespace std;

void swap(int *p, int *q) {
   int tmp;
   tmp = *p;       /* tmp = 3 */
   *p = *q;        /* *p = 7 */
   *q = tmp;       /* *q = 3 */
}

int main() {
   int a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl;
    /* 7  3 is printed */
   return 0;
}
```



Example: Swapping Value

```
#include <iostream>
using namespace std;

void swap(int *p, int *q) {
   int tmp;
   tmp = *p;           /* tmp = 3 */
   *p = *q;            /* *p = 7 */
   *q = tmp;           /* *q = 3 */
}

int main() {
   int a = 3, b = 7;
   swap(&a, &b);
   cout << a << " " << b << endl;
    /* 7  3 is printed */
   return 0;
}
```



Outlines

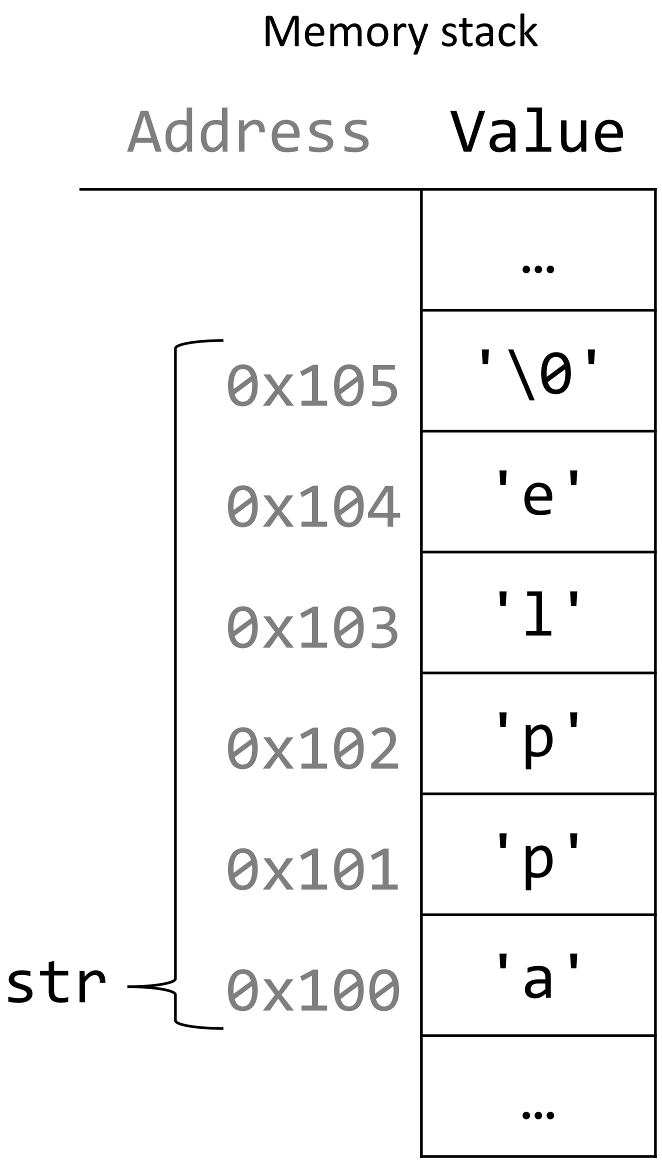
- Memory and variable
- Pointer and its operations
- Pass by pointer
- Array and pointer

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



Memory stack

	Address	Value
		…
	0x105	'\0'
	0x104	'e'
	0x103	'l'
	0x102	'p'
	0x101	'p'
str	0x100	'a'
		…

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

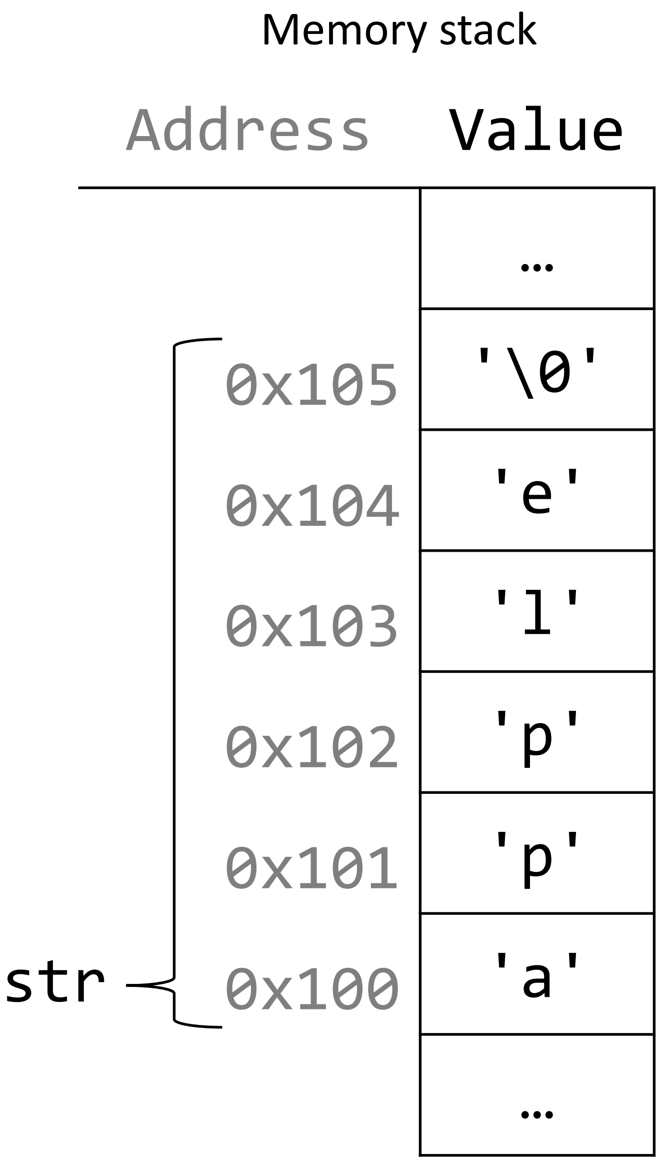
- when we declare an array of characters, continuous memory is allocated on the memory stack to store the contents of the entire array

```
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
```



Memory stack

Address	Value
	…
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
str 0x100	'a'
	…

Array Variable is NOT a Pointer

- Reassignment of array variable is NOT allowed

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

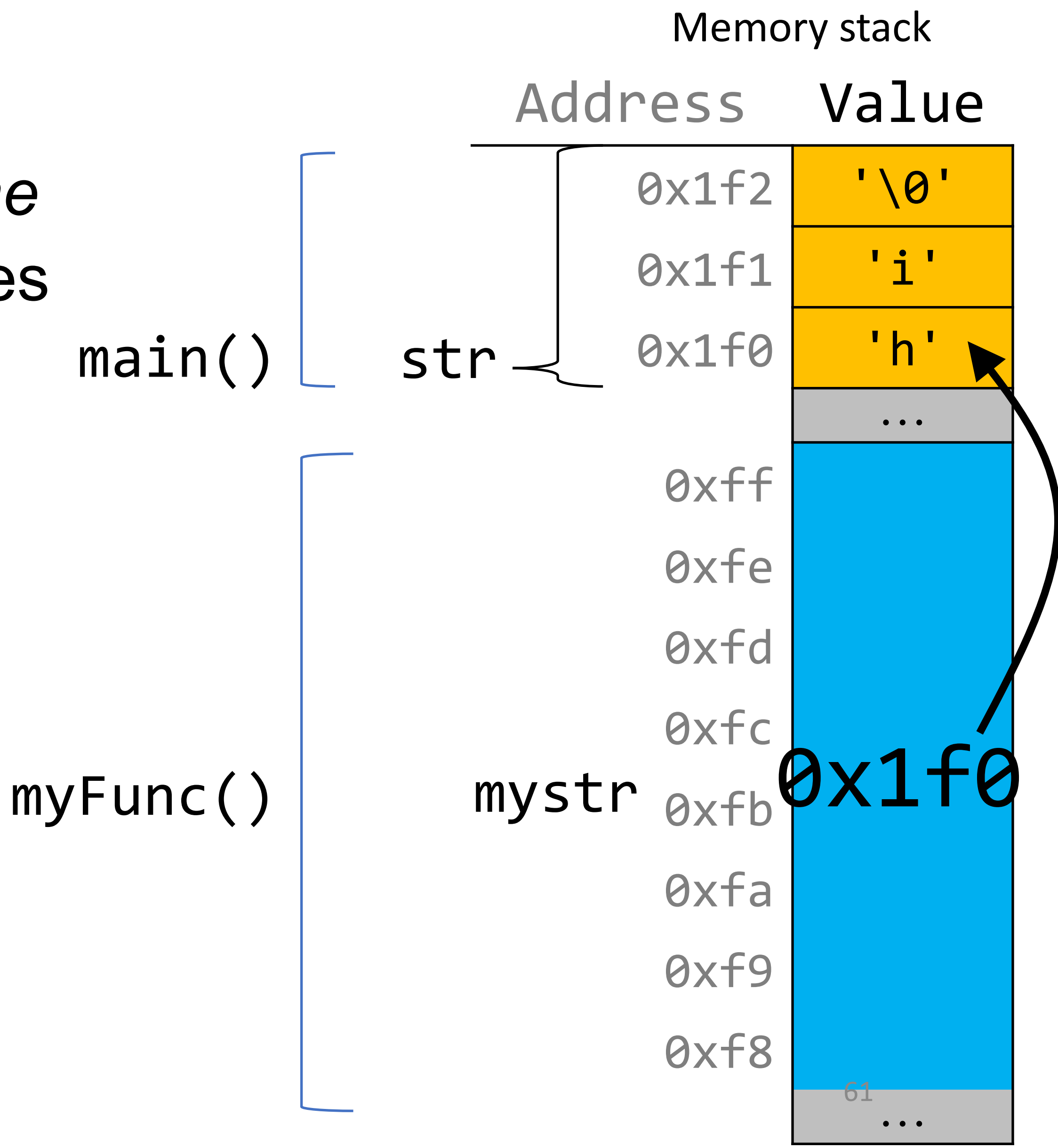
- In comparison, reassignment of pointer is allowed

```
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.

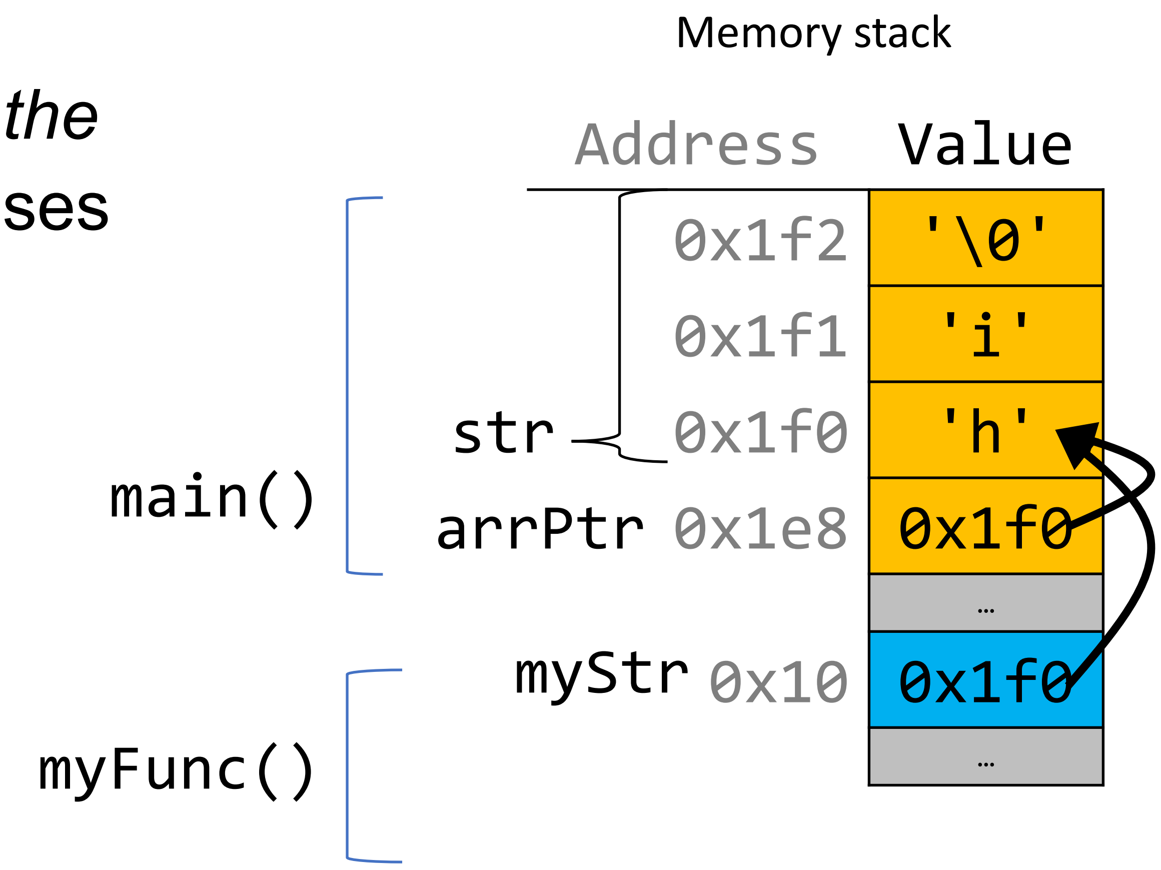
```
void myFunc(char *myStr) {
      ...
}
void main() {
      char str[3];
      strcpy(str, "hi");
      // equivalent
      char *arrPtr = str;
      myFunc(str);
}
```



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
        char *arrPtr = str;
        myFunc(str);
}
```



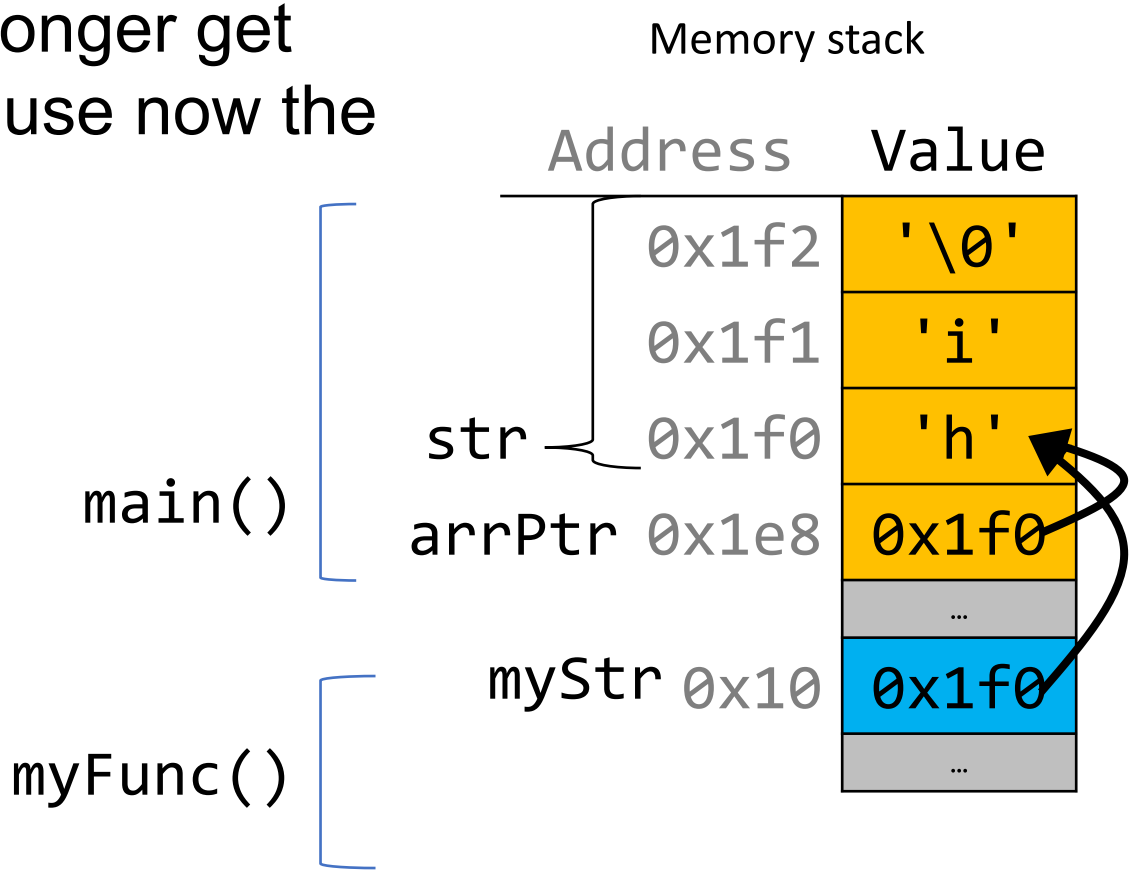
Memory stack

		Address	Value
main()		0x1f2	'\0'
		0x1f1	'i'
	str	0x1f0	'h'
	arrPtr	0x1e8	0x1f0
			…
myFunc()	myStr	0x10	0x1f0
			…

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");
      // equivalent
      cout << sizeof(myStr); // 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