CS2311 Computer Programming

LT09: Pointer I

Computer Science, City University of Hong Kong

Semester B 2022-23

Outlines

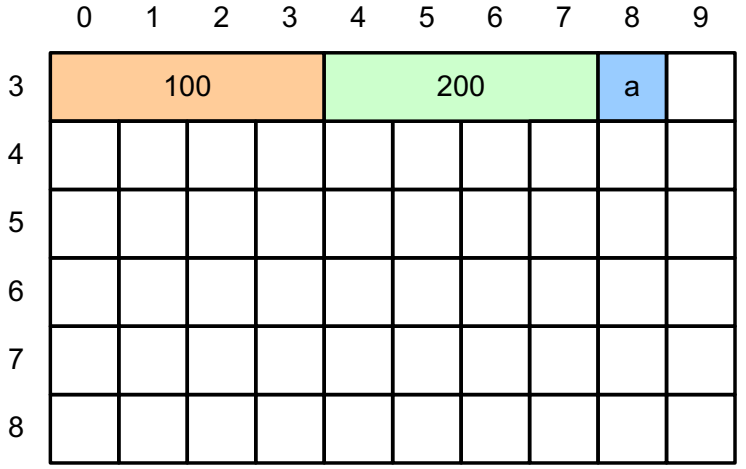
- Recap: variable and memory
- Pointer and its operations
- Pass by pointer
- Array and pointer

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

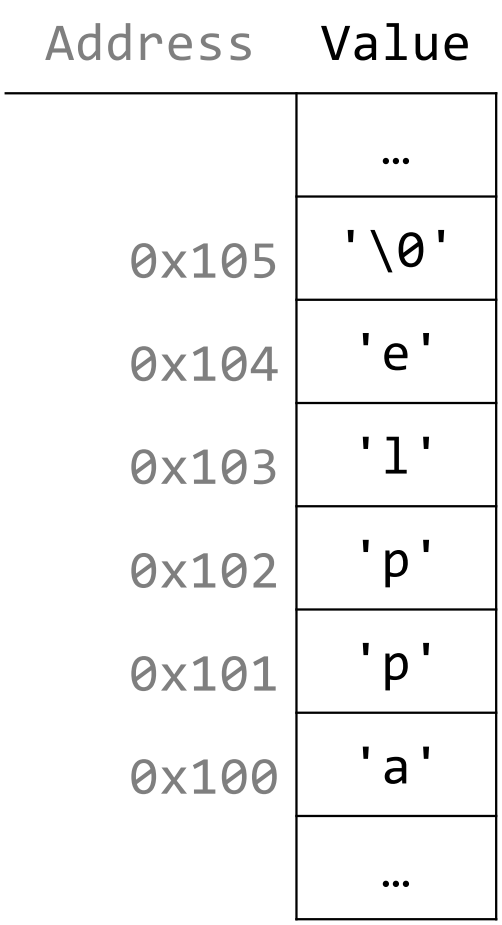
```
void main (){
        int  x;
        int  y;
        char c;
        x = 100;
        y = 200;
        c = 'a';
}
```



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



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 *

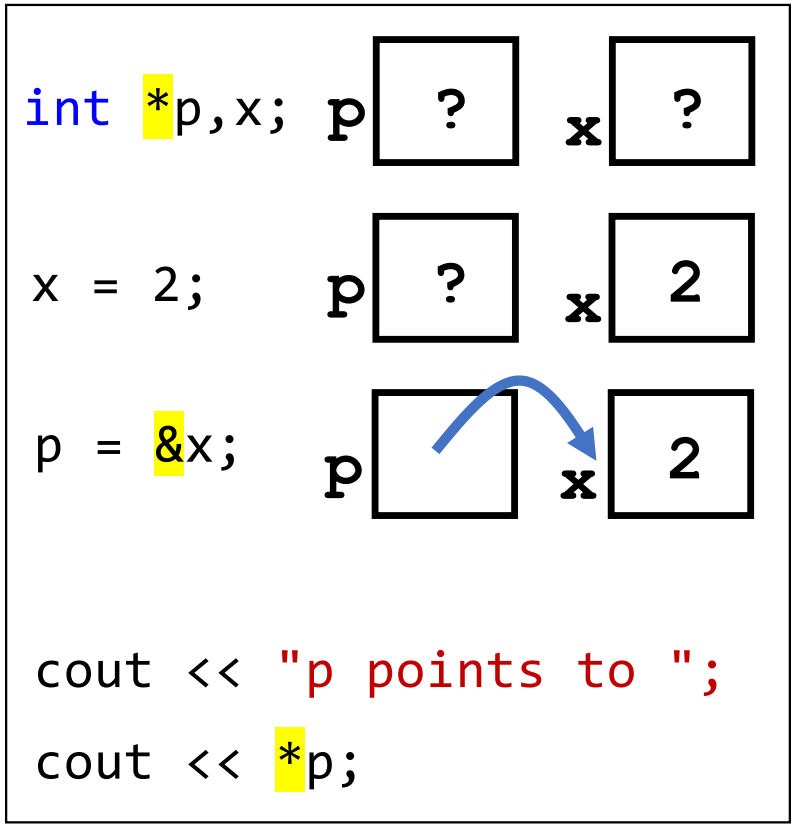
```
int x = 2;
// Make a pointer that stores the address of x
// To declare an int pointer, place a "*" before identifier
// assign address of x to pointer (& is address operator here)
int *xPtr = &x;
// Dereference the pointer to get the value stored in that address
// (* is the dereference operator in this context)
cout << *xPtr;          // prints 2
```

Basic Pointer Operators: & and *

& address operator: get address of a variable

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

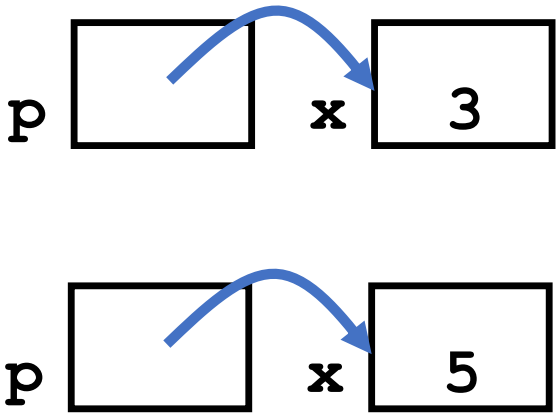
- in declaration (such as int* p), it indicates a *pointer type* (e.g., int *p is a pointer which points to an int variable)
- when it appears in other statements (such as cout << *p), it's a *deference operator* which gets the value of the variable pointed by *p.*



Basic Pointer Operators: & and *

- write a value into memory using dereference operator *
- 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;
```

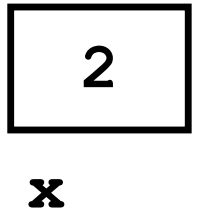
Outlines

- 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.

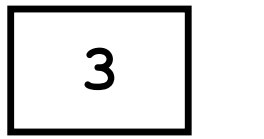
```
void myFunc(int& num) {
      num = 3;
}

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



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;
}
```



x, num

num is an alias name of x.

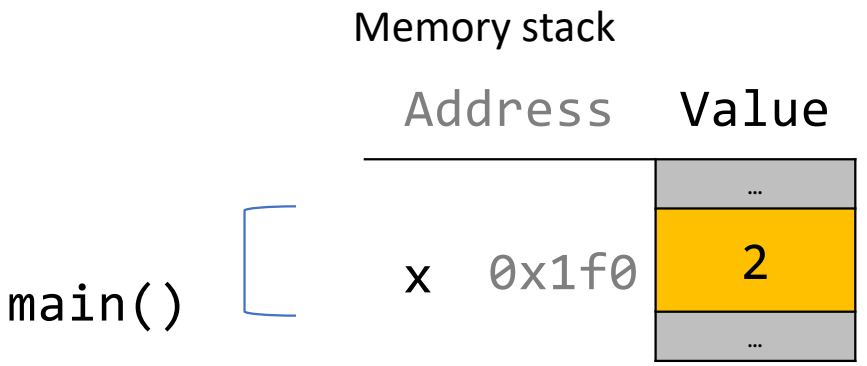
Pass-by-Reference vs Pass-by-Pointer

```
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;
}
```

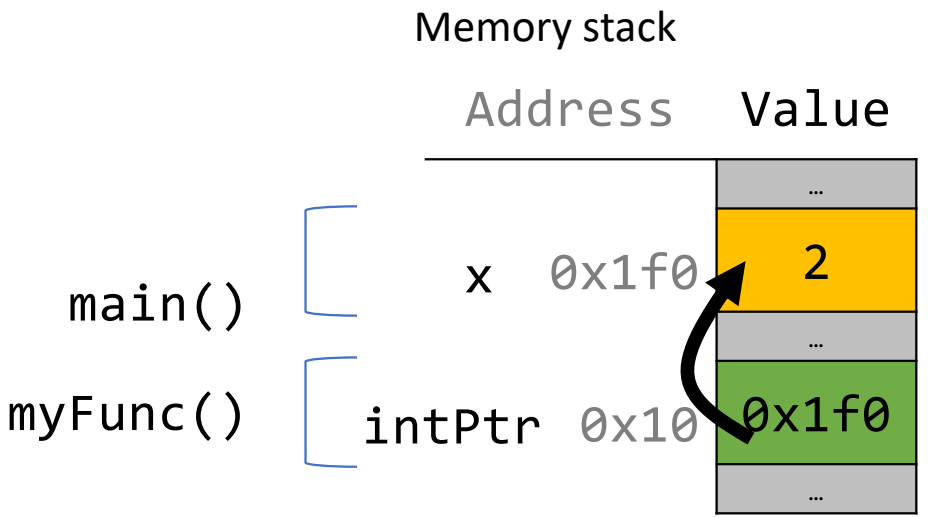
Pass-by-Pointer

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

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



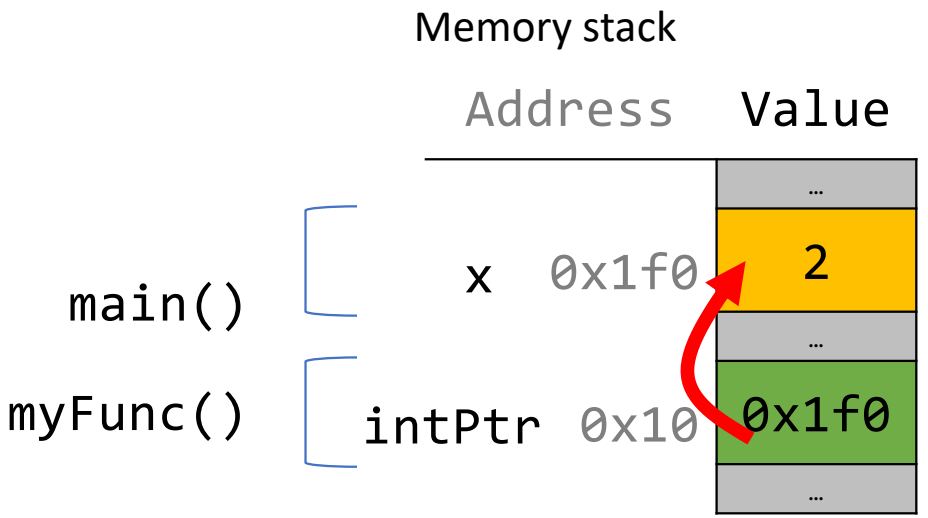
Pass-by-Pointer

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

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



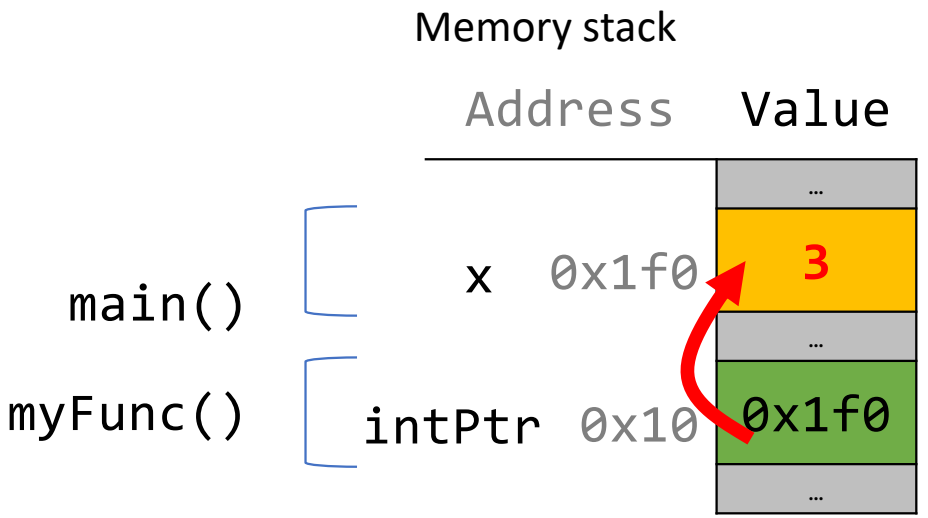
Pass-by-Pointer

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

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



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

Pass-by-Pointer

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

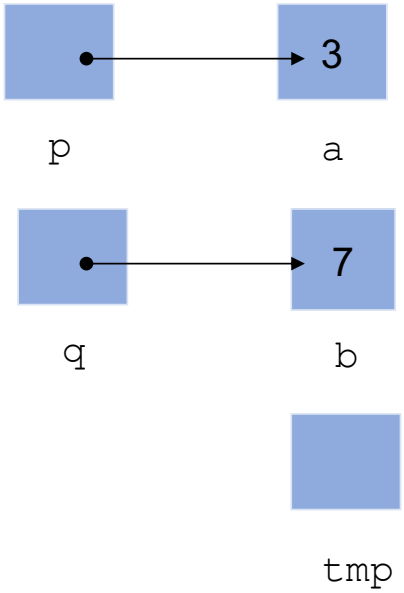
Pass-by-Pointer vs Pass-by-Reference

```
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;
}
```

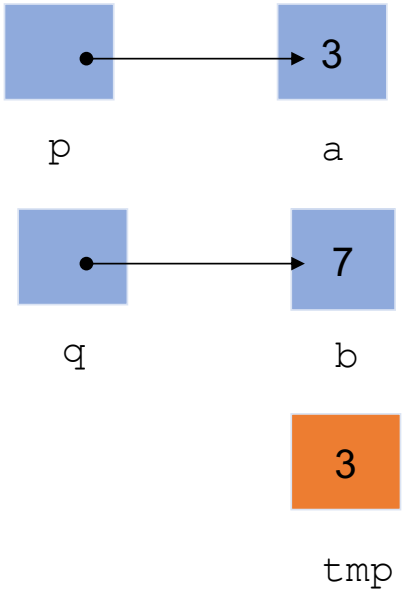
Example: Swapping Value

```
#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;
}
```



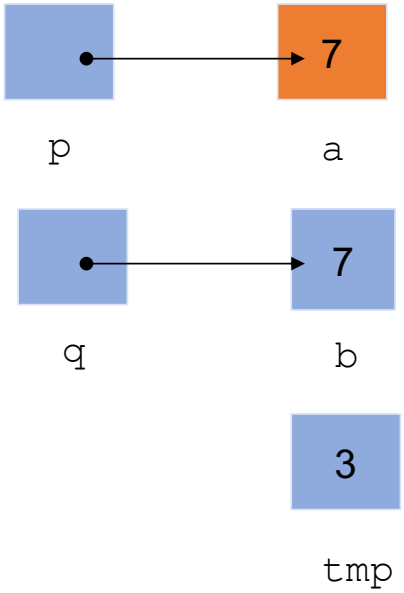
Example: Swapping Value

```
#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;
}
```



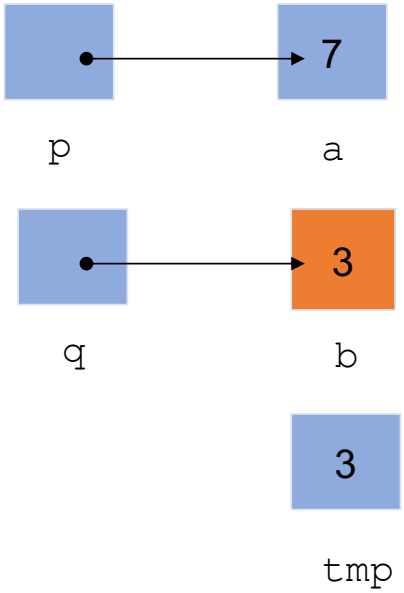
Example: Swapping Value

```
#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

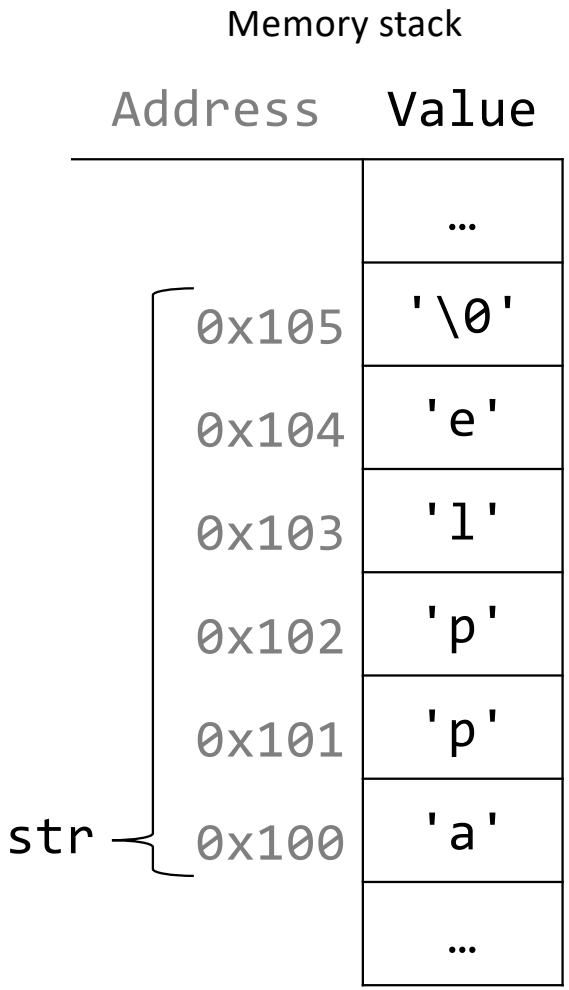
- Memory and variable
- Pointer and its operations
- Pass by pointer
- Array and pointer

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



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"
```

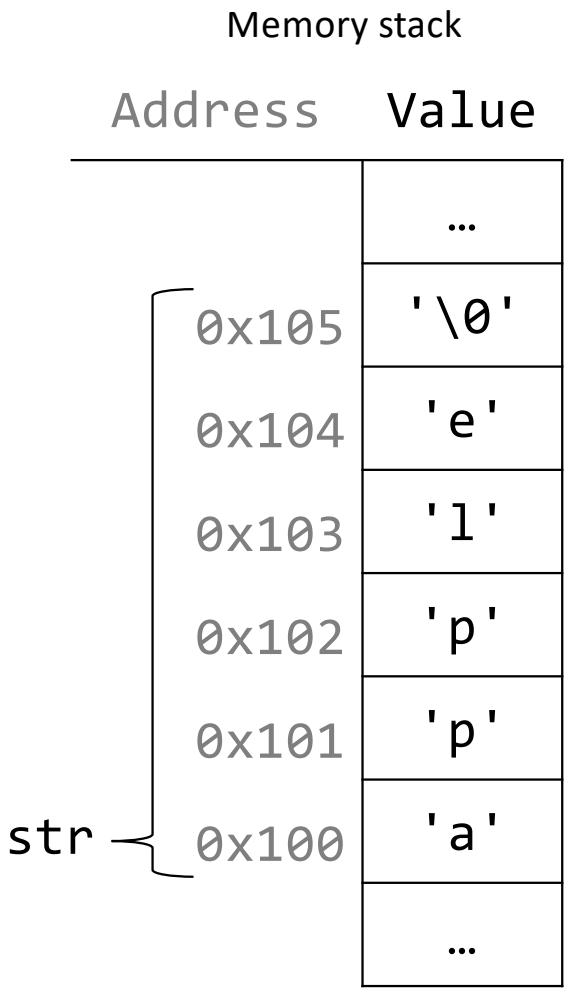
Array Variable is NOT a Pointer

- 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]);
```



Array Variable is NOT a Pointer

- Reassignment of array variable is NOT allowed

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

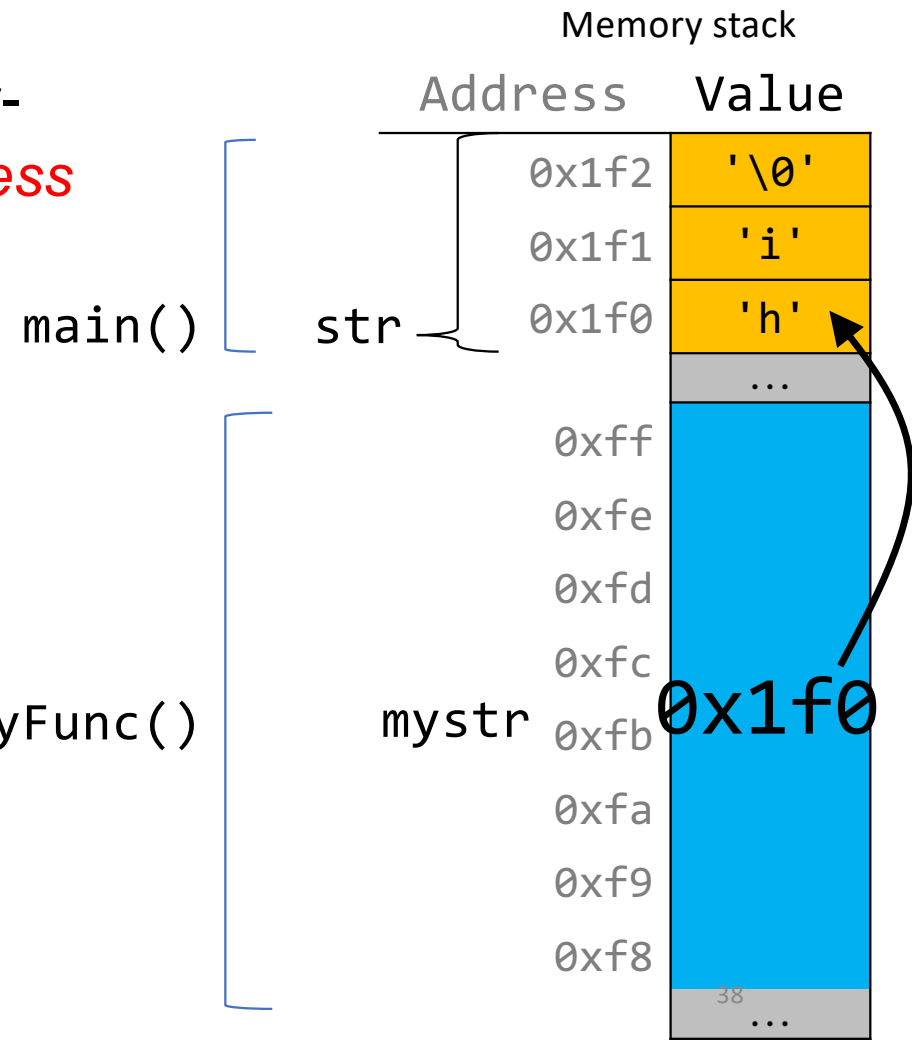
- In comparison, reassignment of pointer is allowed

```
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, C makes a *copy of the address of the first array element and* passes it as a pointer to the function.

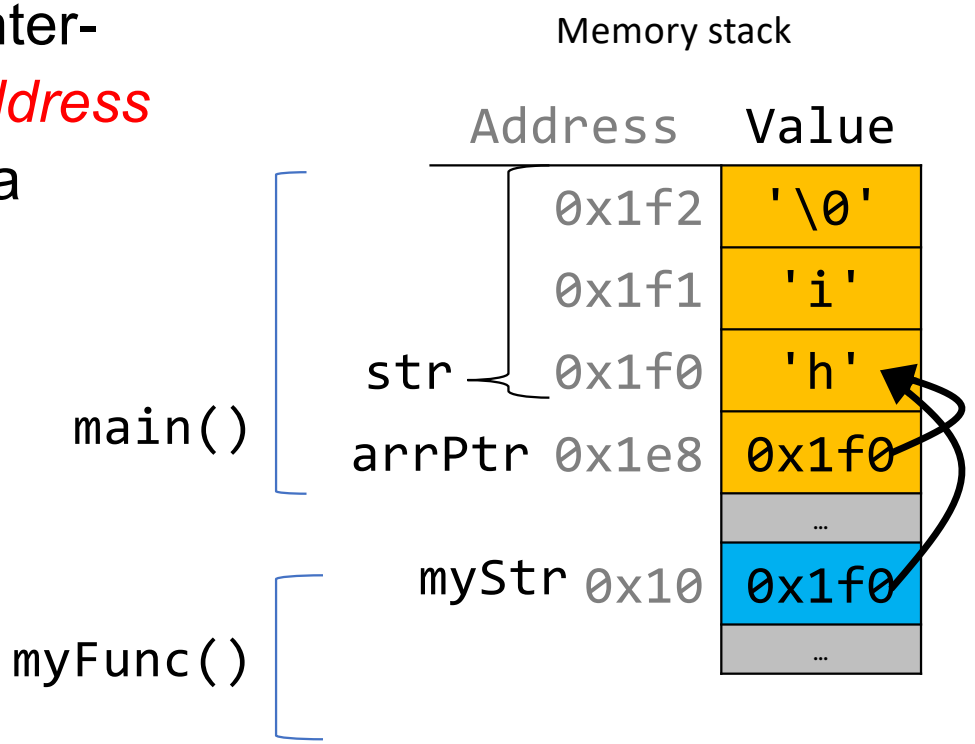
```
void myFunc(char *myStr) {
      ...
}
void main() {
      char str[3];
      strcpy(str, "hi");
      // equivalent
      char *arrPtr = str;
      myFunc(str);
}
```



Arrays as Parameters

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

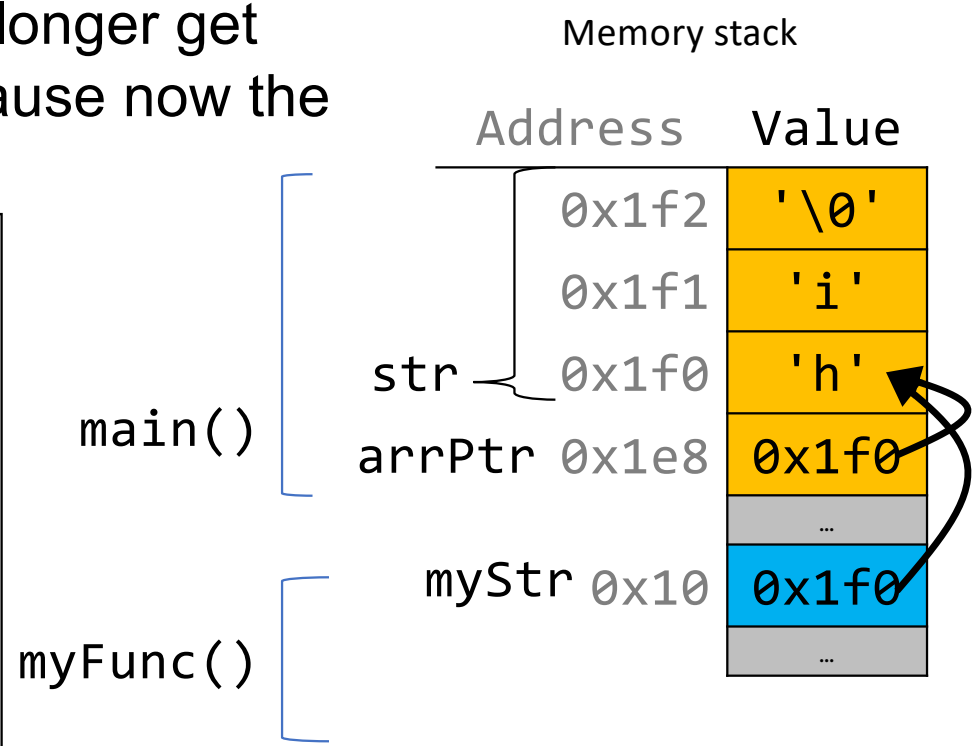
```
void myFunc(char *myStr) {
      ...
}
void main() {
      char str[3];
      strcpy(str, "hi");
      // equivalent
      char *arrPtr = str;
      myFunc(str);
}
```



Arrays as Parameters

- 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