CS2310 Computer Programming

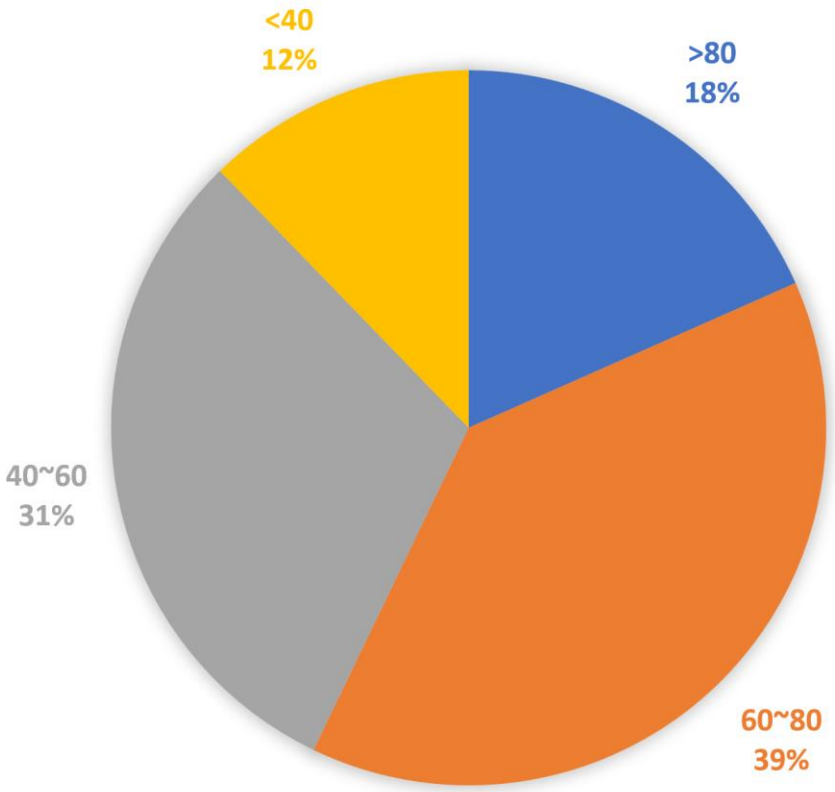
LT09: Pointer I

Computer Science, City University of Hong Kong

Semester A 2023-24

Mid-term Statistics

- Average: 59.96 (out of 100)
- Median: 62
- Highest: 93
- Distribution:
  - >80: 9 students;
  - 60~80: 19 students;
  - 40~60: 15 students;
  - ≤40: 6 students;



Mid-term Enquiry

- Any questions related to Midterm grade should be directed to

xinyanli4-c@my.cityu.edu.hk, and

longxwang4-c@my.cityu.edu.hk and

CC me at

yf.du@cityu.edu.hk

Outlines

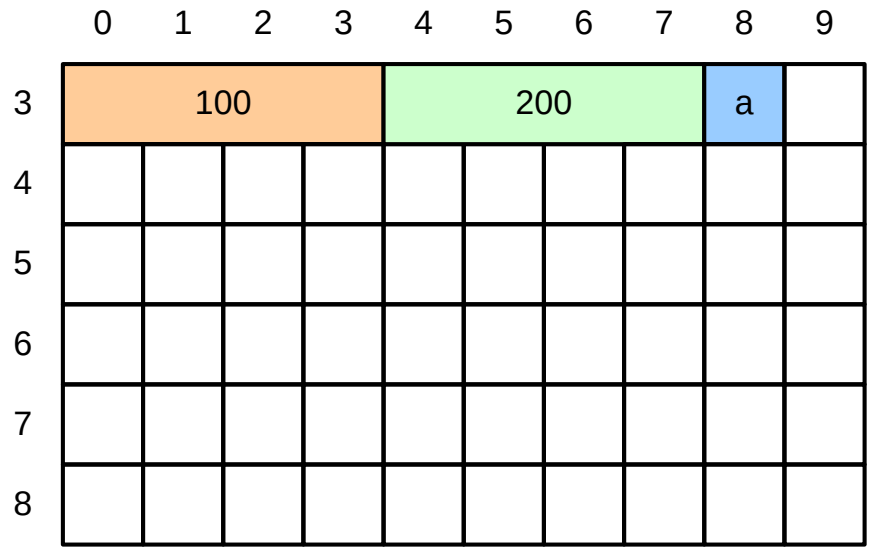
- Recap: variable and memory
- Pointer and its operations
- Pass by pointer
- Array and pointer

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

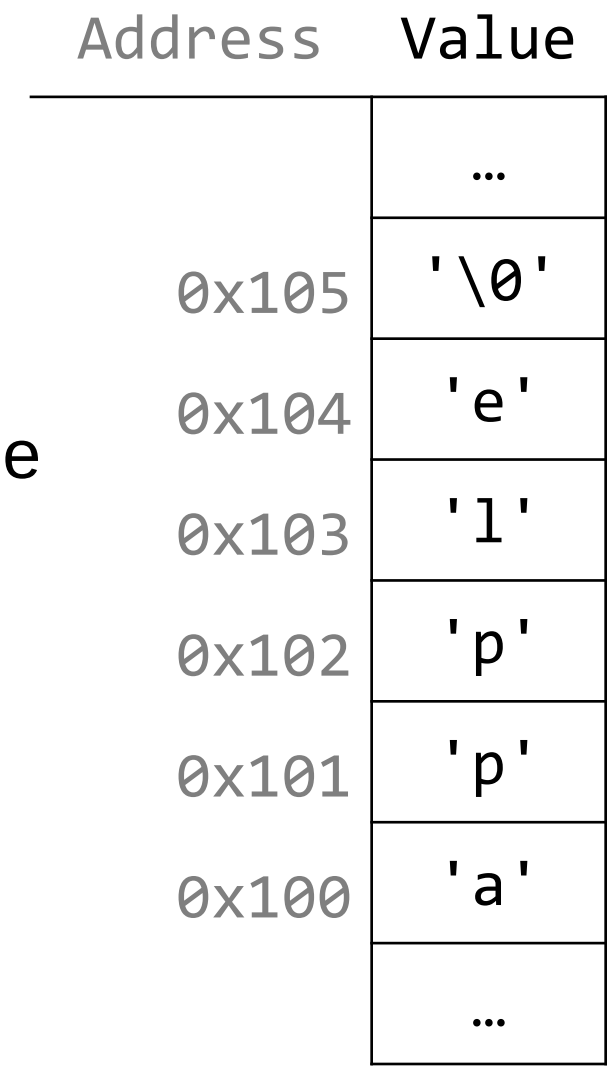
```
void main (){
        int  x;
        int  y;
        char c;
        x = 100;
        y = 200;
        c = 'a';
}
```



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

Variable type	int x;	float x;	double x;	char x;
Pointer type	int* Pointx;	float* Pointx;	double* Pointx;	char* Pointx;

Outlines

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

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 go to that address
// (* is the dereference operator in this context)
cout << *xPtr;   // prints 2
```

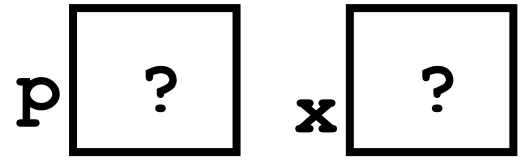
Basic Pointer Operators: & and *

& address operator: get address of a variable

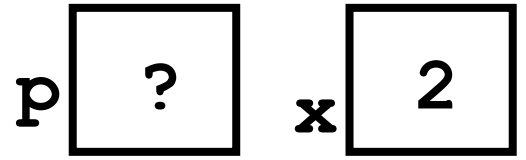
***** is used in **TWO** different ways

- 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.*

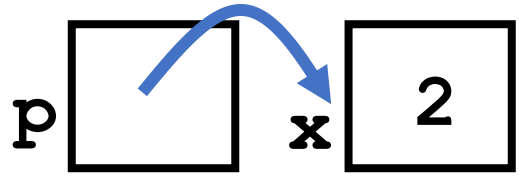
```
int *p,x;
```




```
x = 2;
```




```
p = &x;
```



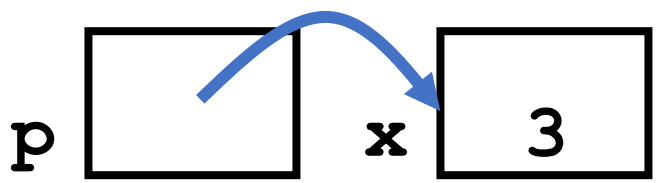

```
cout << "p points to ";
cout << *p;
```

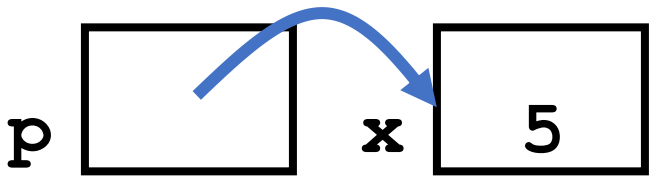
Basic Pointer Operators: & and *

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*;

Exercise: What're the Errors?

```
int x = 3;

char c = 'a';

char *ptr;

ptr = &x;

ptr = c;


ptr = &c;
```

Exercise: What're the 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

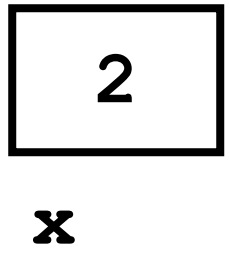
& sign is called reference declarator in this context.

```
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.

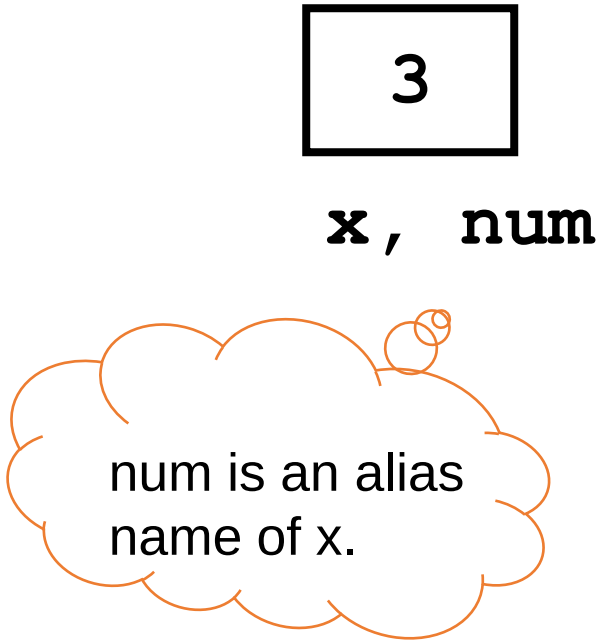
```
void myFunc(int& num) {
      num = 3;
}

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



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;
}
```



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;
}
```

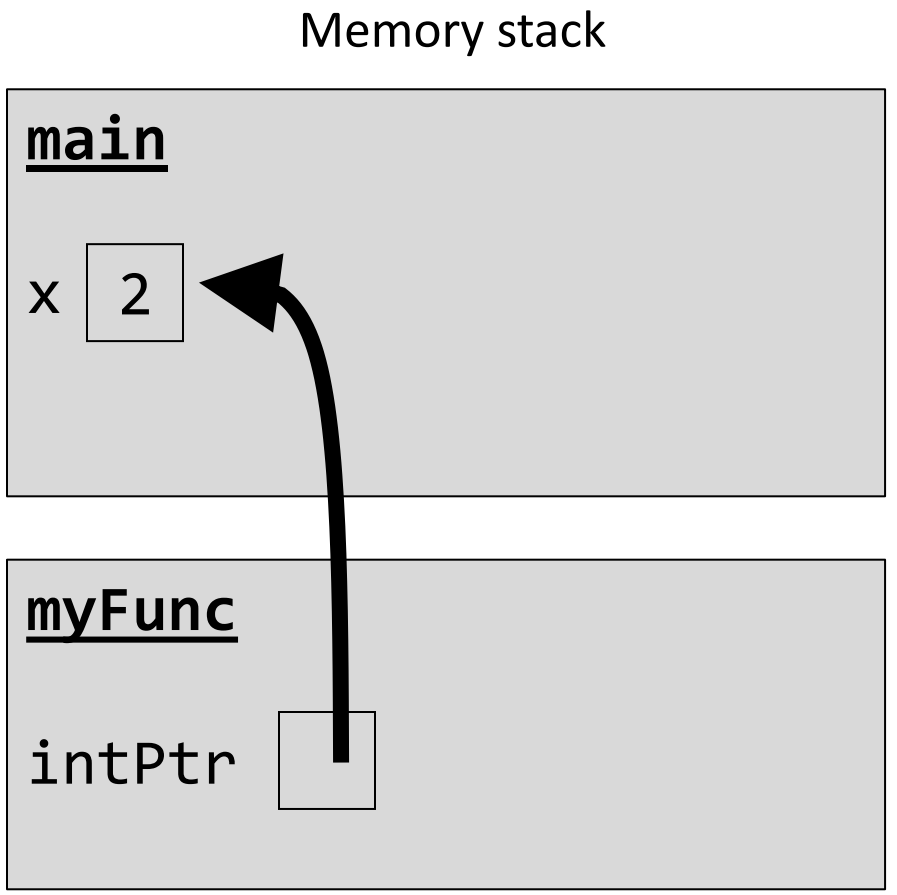
Pass by Pointer

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

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

Memory stack

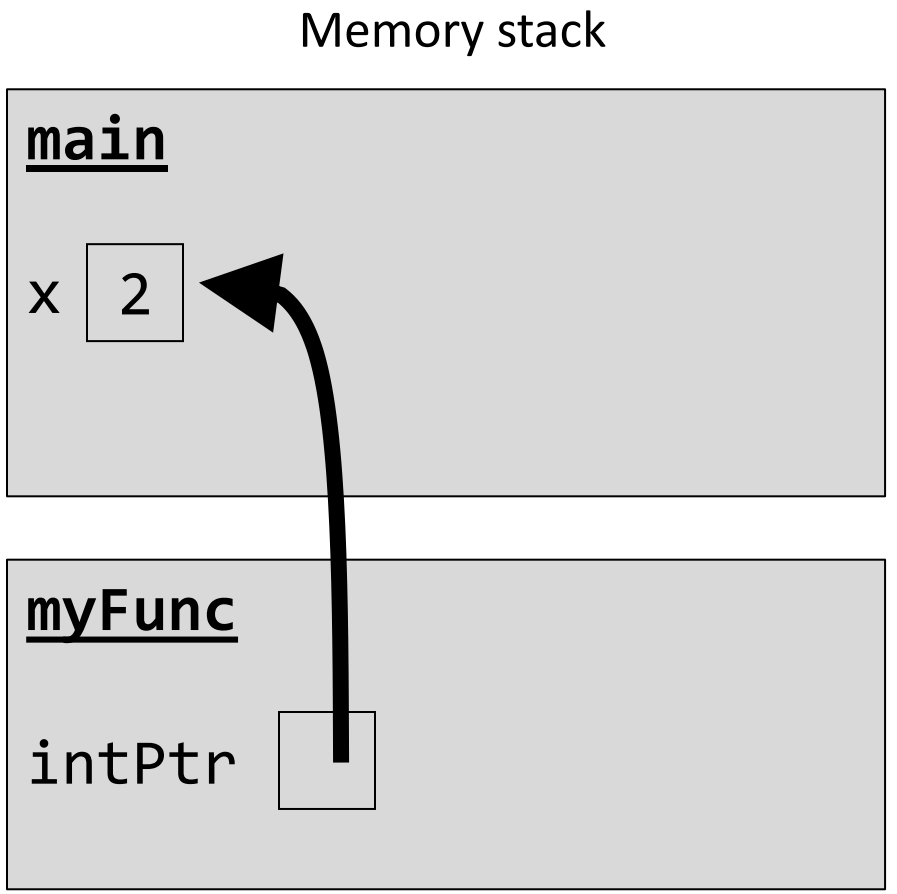
Pass by Pointer

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

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



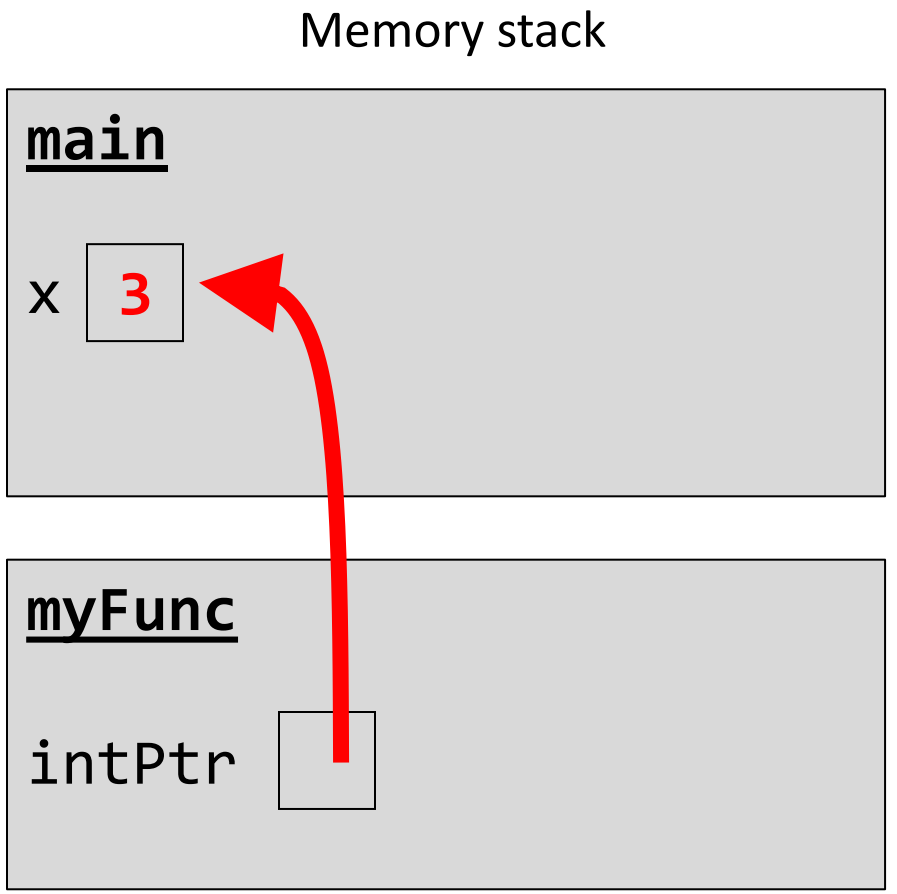
Pass by Pointer

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

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



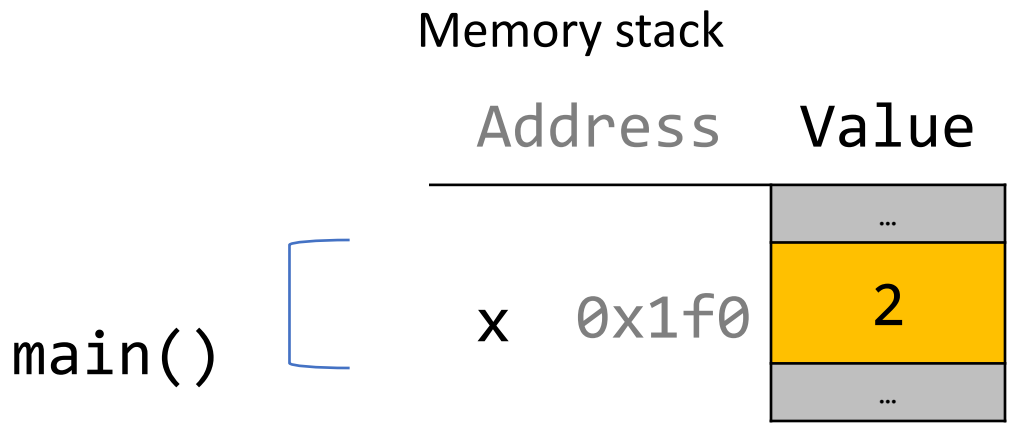
Pass by Pointer

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

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



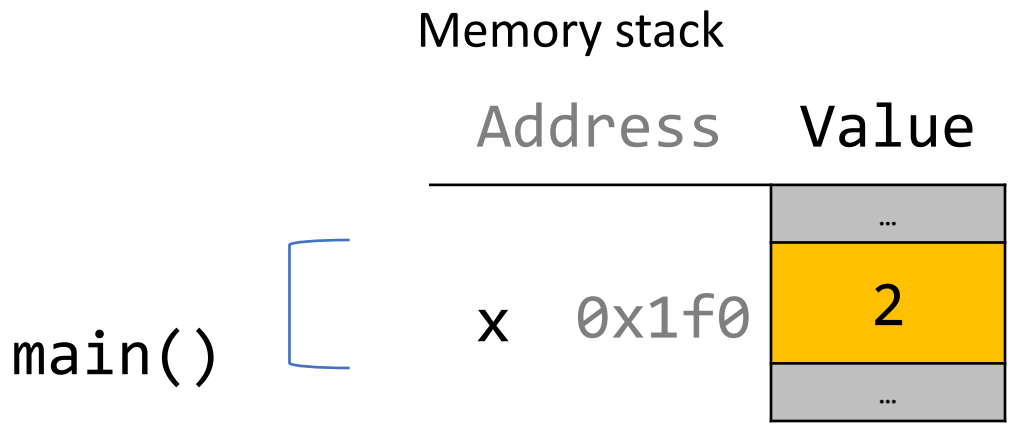
Pass-by-Pointer

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

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



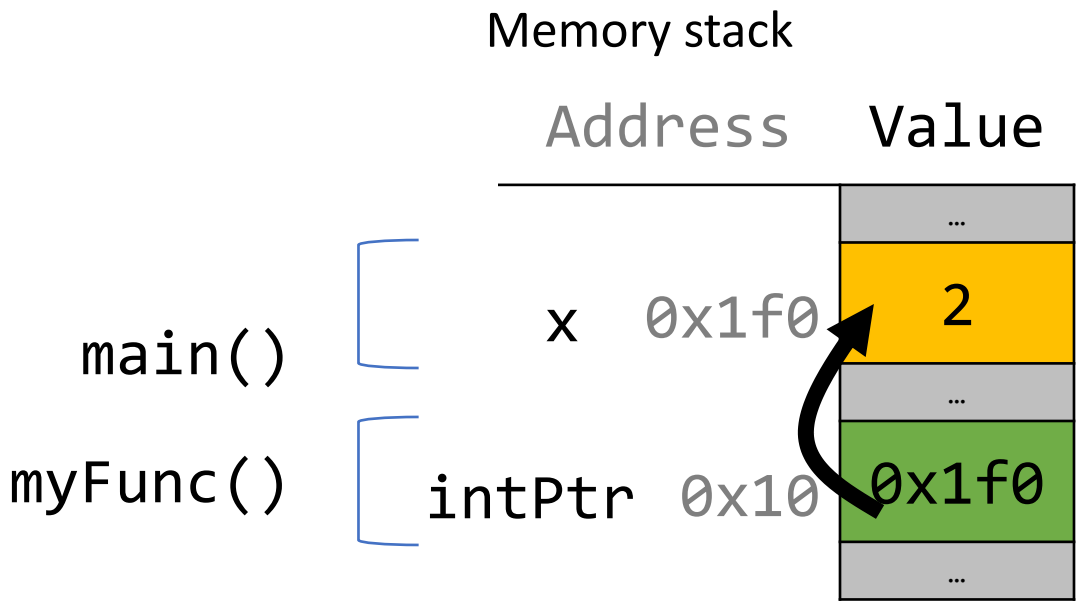
Pass-by-Pointer

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

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



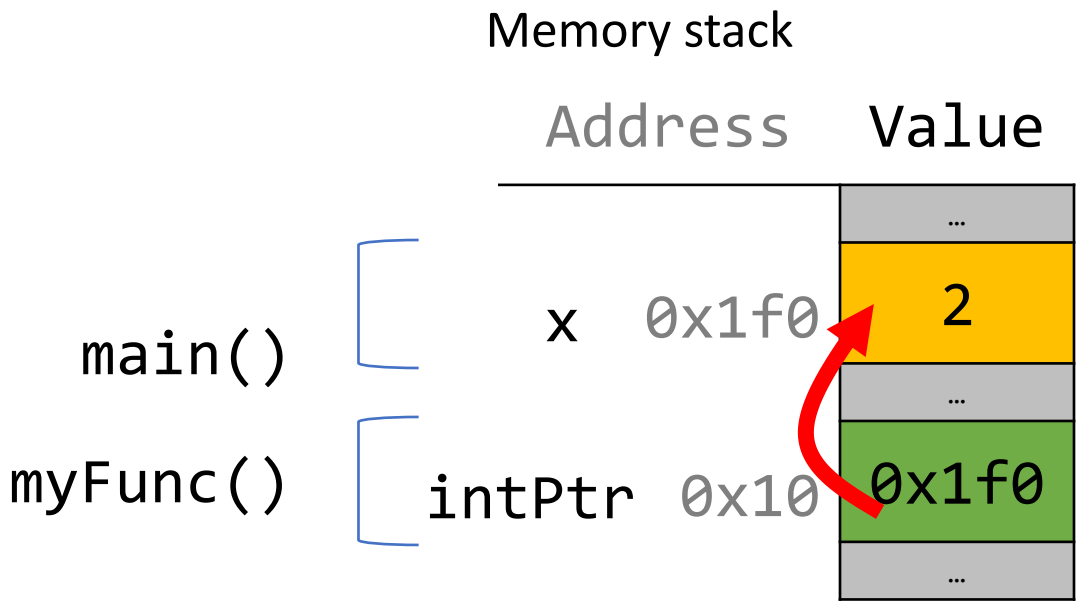
Pass-by-Pointer

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

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



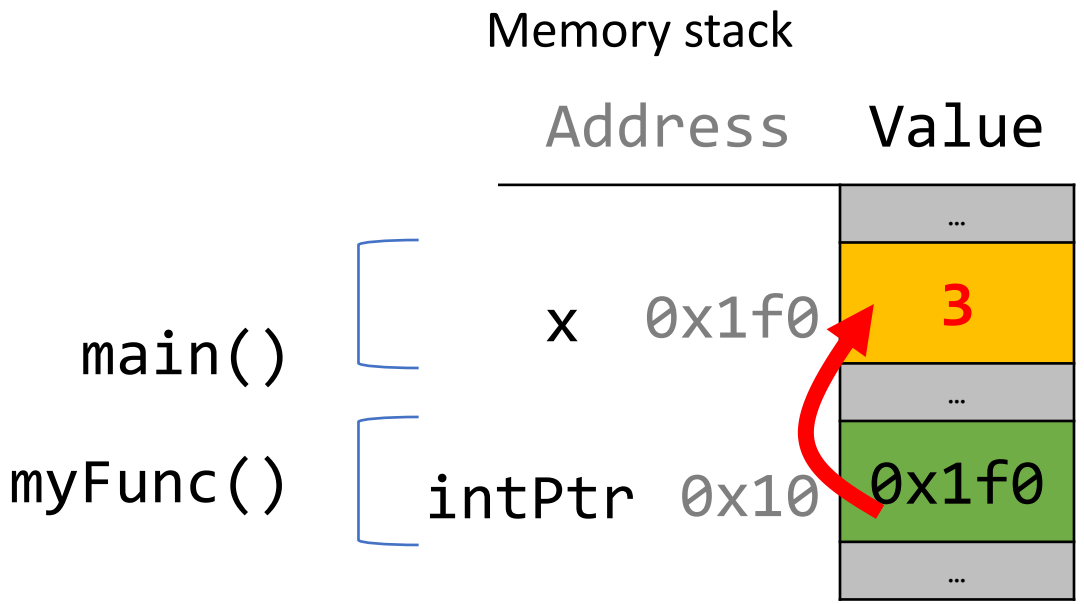
Pass-by-Pointer

```
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;
}
```



Call by Value and Call by Pointer

- In **call by value**, only a single value can be returned using a return statement
  - The inputs will not be affected, as a copy is passed into the function
- In **call by pointer**, the argument(s) can be a pointer which points to the variable(s) in the caller function
  - More than one variables can be updated, achieving the effect of returning multiple values

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;
}
```

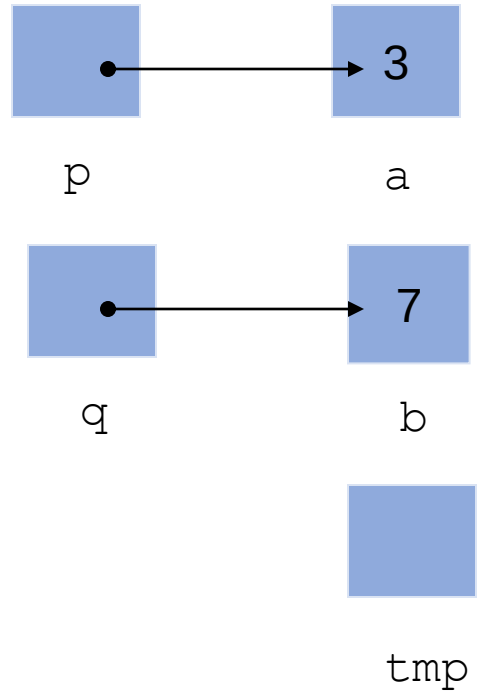
Pass-by-Pointer vs Pass-by-Reference

```
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;
}
```

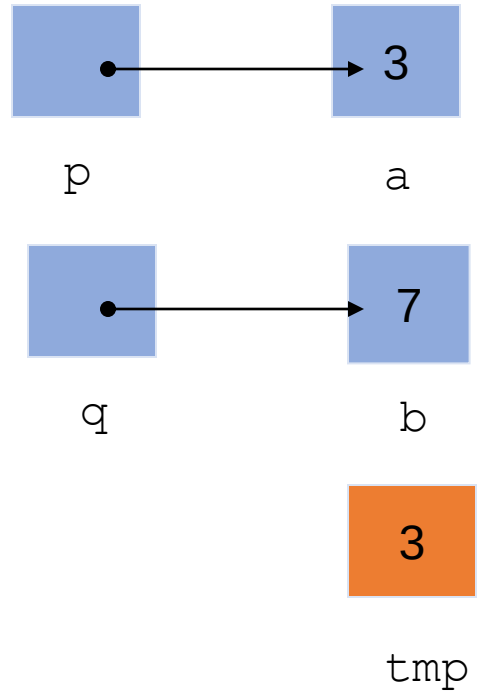
Example: Swapping Value

```
#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;
}
```



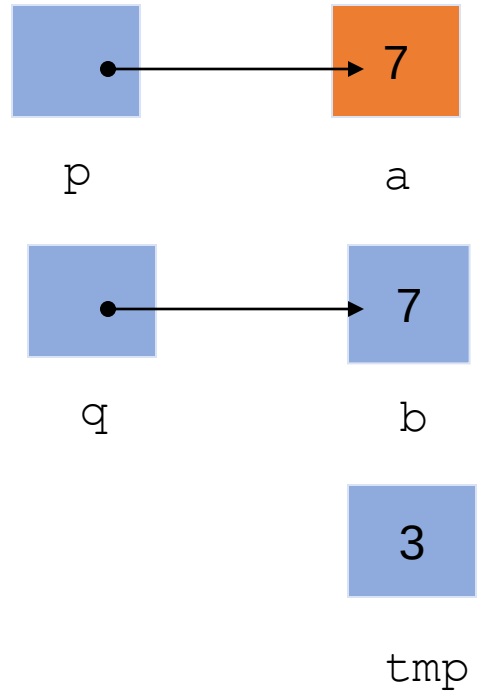
Example: Swapping Value

```
#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;
}
```



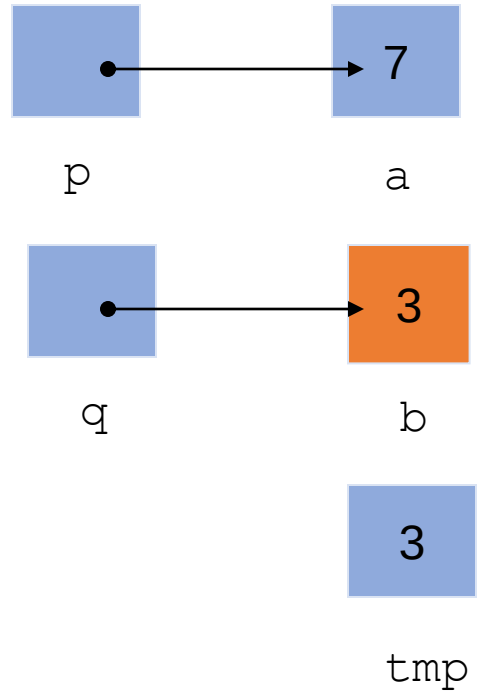
Example: Swapping Value

```
#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

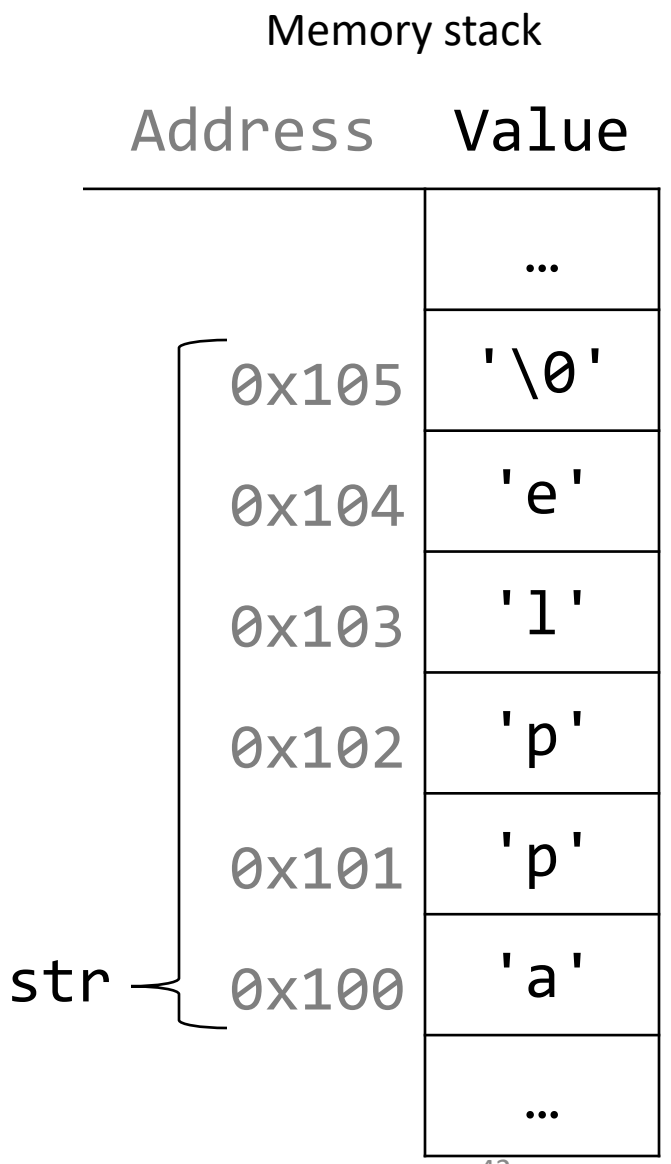
Arrays

When we declare an array of characters, contiguous 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**) can refer to the entire array contents. In fact, **sizeof** returns the size of the entire array!

```
int arrayBytes = sizeof(str);     // 6
```



Arrays

An array variable refers to an entire block of memory. We cannot reassign an existing array to be equal to a new array.

```
int nums[] = {1, 2, 3};
int nums2[] = {4, 5, 6, 7};

nums = nums2; // not allowed!
```

An array's **size** cannot be changed once we create it; we must create another new array instead.

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.

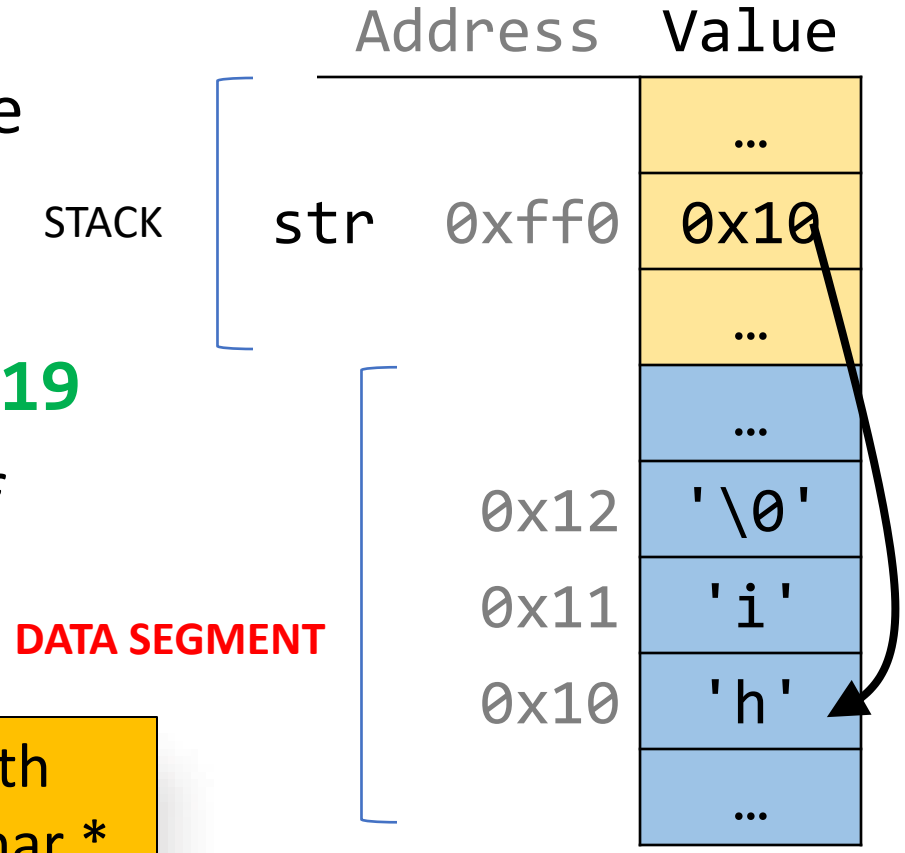
```
char str[] = "Hello World";

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

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

char *

When we declare a char pointer equal to a string literal, the characters are *not* stored on the stack. Instead, they are stored in a special area of memory called the "**data segment**". We ***CANNOT*** *modify memory in this segment*.

```
char *str = "hi"; // Disabled in MSVC2019
```

The pointer variable (e.g. **str**) refers to the *address of the first character of the string in the data segment*.



This applies only to creating *new* strings with char *. This does **not** apply for making a char * that points to an existing stack string.

char *

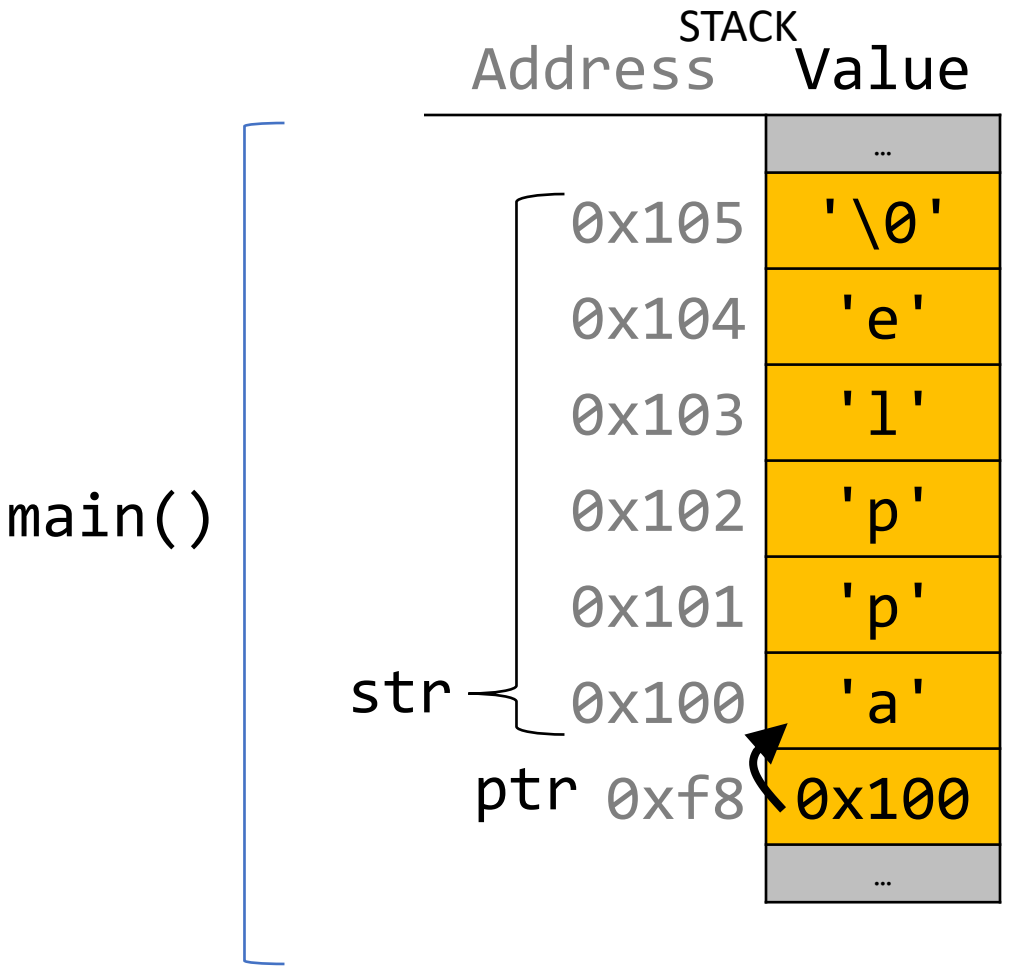
A **char** * variable essentially refers to a single character. We can reassign an existing **char** * pointer to be equal to another **char** * pointer.

```
const char *str = "apple";          // e.g. 0xfff0
char *str2 = "apple 2";       // e.g. 0xfe0
str = str2;    // ok!  Both store address 0xfe0
```

Arrays and Pointers

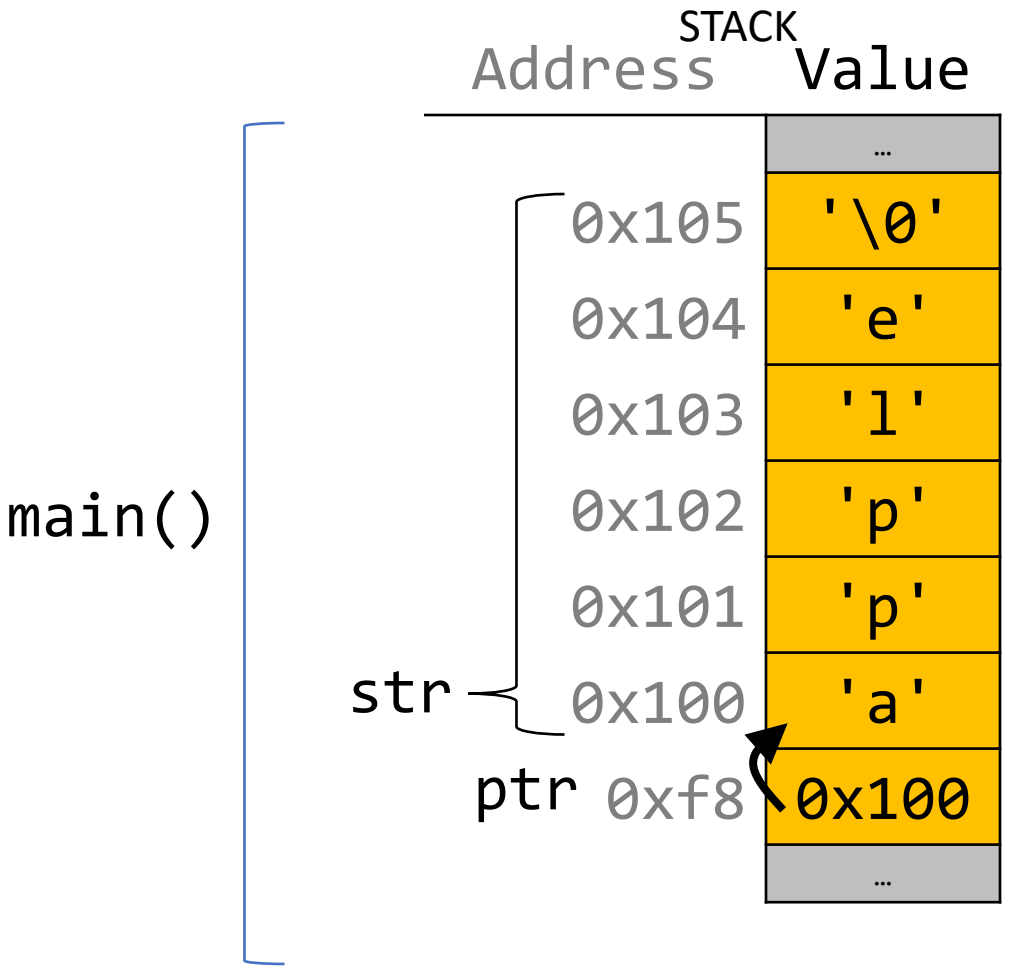
We can also make a pointer equal to an array; it will point to the first element in that array.

```
int main() {
      char str[6];
      strcpy(str, "apple");
      char *ptr = str;
      return 0;
}
```



Arrays and Pointers

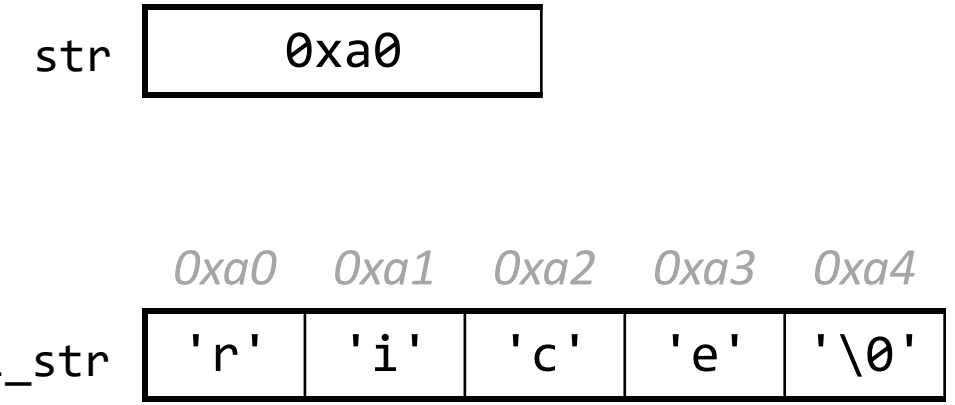
```
int main() {
    char str[6];
    strcpy(str, "apple");
    char *ptr = str;

    // equivalent
    char *ptr = &str[0];

    // Not valid on most platforms
    char *ptr = &str;

    return 0;
}
```



Arrays as Parameters

How do you think the parameter `str` is being represented?

```
void myFunc(char *str) {
      ...
}

int main() {
      char local_str[5];
      strcpy(local_str, "rice");
      myFunc(local_str);
      return 0;
}
```



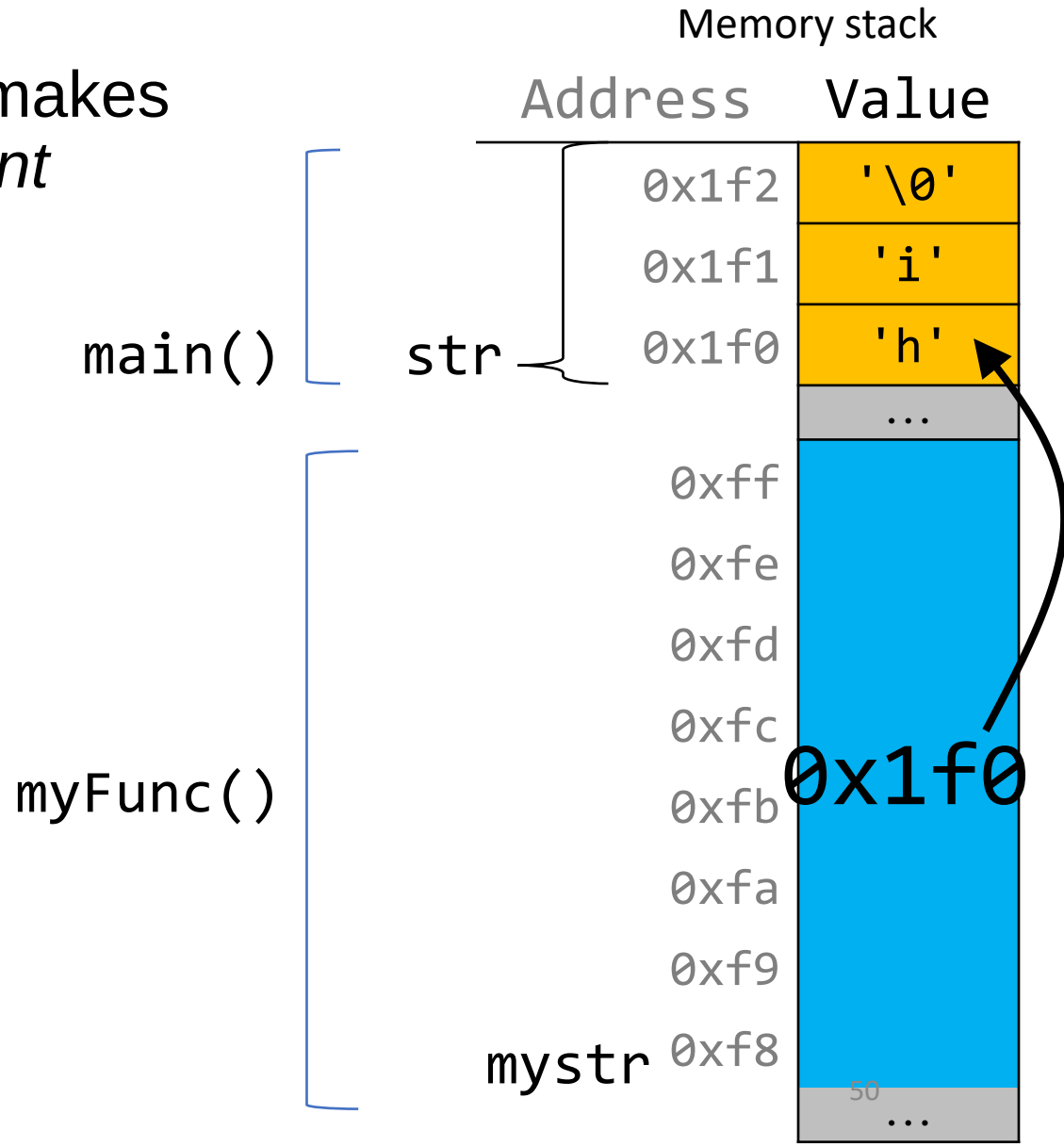
A. A copy of the array `local_str`

B. A pointer containing an address to the first element in `local_str`

Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element and* passes it (a pointer) to the function.

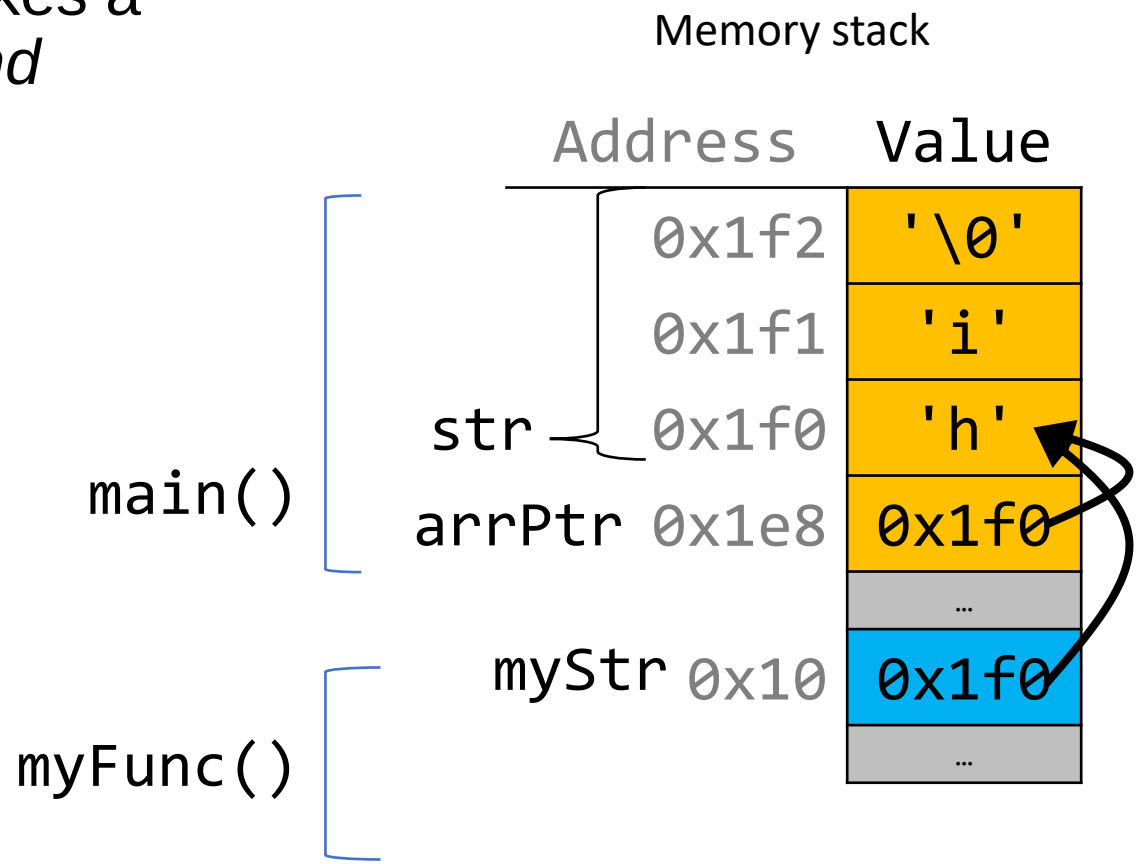
```
void myFunc(char *myStr) {
      ...
}

void main() {
      char str[3];
      strcpy(str, "hi");
      myFunc(str);
}
```



Arrays as Parameters

When you pass an **array** as a parameter, C makes a *copy of the address of the first array element and* passes it (a pointer) to the function.

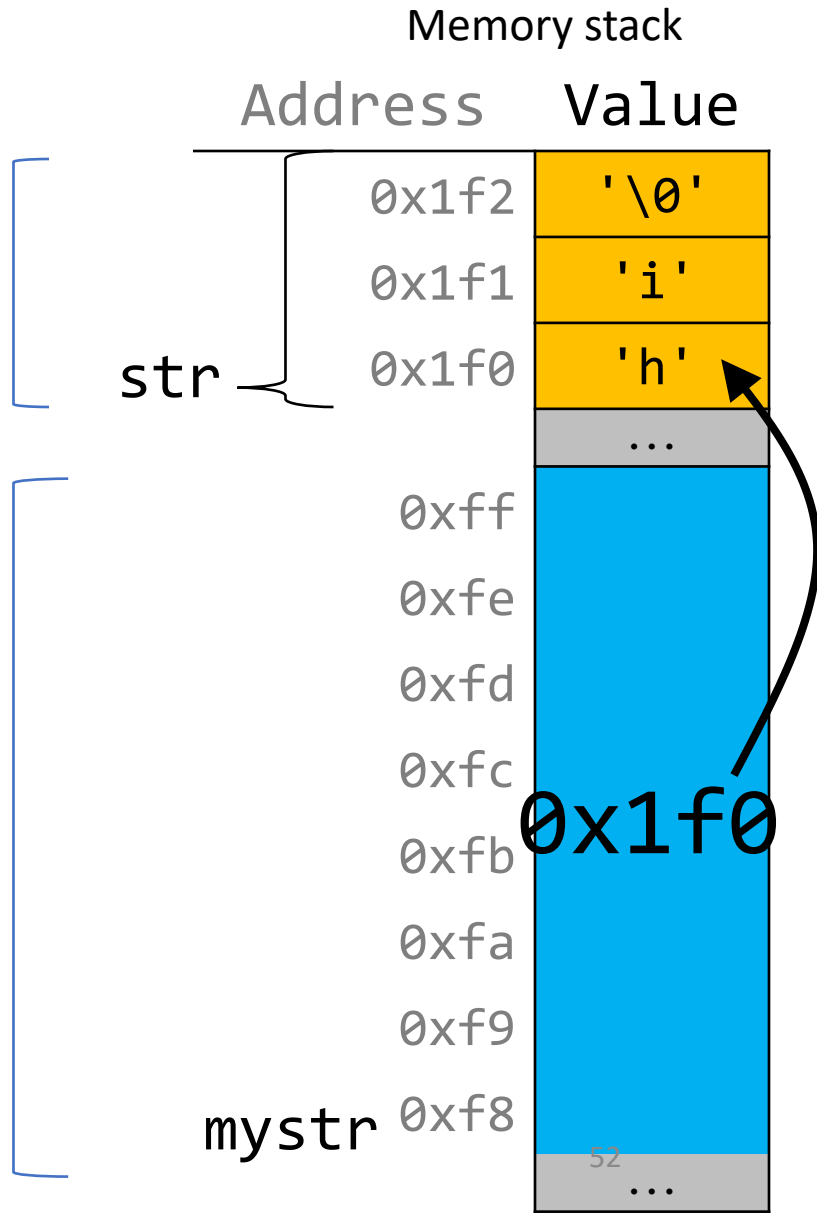
```
void myFunc(char *myStr) {
      ...
}

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



Arrays as Parameters

This also means we can no longer get the full size of the array using **sizeof**, because now it is just a pointer. But for cstring, we can still call **strlen**.

```
void myFunc(char *myStr) {
      int size = sizeof(myStr); // 4 or 8
      int len = strlen(myStr); // 2
}
void main() {
      char str[3];
      strcpy(str, "hi");
      int size = 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
  - Print it out
  - Use string functions
  - But under the hood they are represented differently!
- **Takeaway:** **We create strings as char[], pass them around as char ***

Arrays vs. Pointers

- 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

Summary

- * can be used to declare a pointer
- * can also be a dereference operator
- & is often used as an address operator
- & can also be a reference declarator (valid syntax of C++)
- Array and pointers
- Draw **memory diagrams**!
  - Pointers store addresses. Make up addresses if it helps your mental model.