C++ Classes Review

Additional Resources:

Sean Szumlanski's awesome CS106B lecture notes

Cynthia Bailey + Julie Zelenski's awesome CS106B slides

Based on slides and notes created by John Ousterhout, Jerry Cain, Chris Gregg, and others.

Overview

- What are classes?
- Defining a Class
- Instance Variables
- Constructor
- Methods
- Destructor

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Overview

- **What are classes?**
- Defining a Class
- Instance Variables
- Constructor
- Methods
- Destructor

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

What are classes?

A **class** lets you define your own **custom variable type**.

- You specify *what data is inside it* (instance variables), *what it can do* (public methods), and *how you create one* (constructor)
- defined across two files: a **header** file and an **implementation** file
  - Header file (.h/.hh): contains the *interface* – an outline of what the type can do, but not the implementation
  - Implementation file (.cc / .cpp): contains all method implementations - internal code
- Benefit: abstract away complexity of type into separate files
- Client (code that uses this variable type) vs. implementer (implementation of class). Clients create *instances* of this class.

Example: Bank Account

```
BankAccount ba1;
ba1.deposit(2.00);
ba1.withdraw(1.50);
cout << "Balance for first account is "
     << ba1.getBalance() << endl;

BankAccount ba2;
ba2.deposit(60.00);
ba2.withdraw(5.00);
ba2.withdraw(5.00);
cout << "Balance for second account is "
     << ba2.getBalance() << endl;
```

Based on an example courtesy of Cynthia Bailey + Julie Zelenski's awesome CS106B slides

Example: Bank Account

We would like the following client functionality for the **BankAccount** type:

- When a bank account is created, it should start with a balance of 0.
- **void deposit(amount)**: this should deposit the specified amount. It does nothing if the amount is negative.
- **void withdraw(amount)**: this should withdraw the specified amount. It does nothing if the amount exceeds the balance.
- **double getBalance()**: this should return the balance in the account.

Overall: a bank account encapsulates logic about its balance

Overview

- What are classes?
- **Defining a Class**
- Instance Variables
- Constructor
- Methods
- Destructor

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Defining a Class – Header File

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here

private:
    // private internal-only functionality/data goes here
};
```

Tells the compiler "if you see this file more than once while compiling, ignore it after the first time" (so it doesn't think you're trying to define things more than once).

Defining a Class – Header File

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here

private:
    // private internal-only functionality/data goes here
};
```

Rule of thumb: default to private unless there is a need to make something public. (don’t want client to be able to do things it shouldn’t)

Public vs. Private

```
int main() {
    vector<int> nums;

    nums.push_back(10);
    nums.push_back(15);
    nums.push_back(33);

    // THIS WOULD BE SO BAD
    nums.size = 1;

    ...
```

Example courtesy of Sean Szumlanski's awesome CS106B lecture notes

Overview

- What are classes?
- Defining a Class
- **Instance Variables**
- Constructor
- Methods
- Destructor

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Instance Variables / Fields / Member Variables

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here

private:
    // private internal-only functionality/data goes here
    // instance variables
    double balance;
};
```

Every **BankAccount** instance will have its own copy of any instance variables. We declare instance variables in the header file, but usually initialize them in the CC file in the constructor.

Overview

- What are classes?
- Defining a Class
- Instance Variables
- **Constructor**
- Methods
- Destructor

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Constructor

A class’s **constructor** is called when a new variable of that type is created.

```
// client code – this calls the constructor
BankAccount ba1;
// equivalent to
BankAccount ba1();
```

Constructor

A class’s **constructor** is called when a new variable of that type is created.

```
// client code – this calls the constructor
BankAccount *ba = new BankAccount;
// equivalent to
BankAccount *ba = new BankAccount();
```

You do not need to define a constructor in your class if you don’t need one (if you don’t, C++ will just pretend you defined a constructor that is empty).

Constructor

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here
    // constructor
    BankAccount();

private:
    // private internal-only functionality/data goes here
    double balance;
};
```

Constructor

A constructor should perform any necessary initialization so the variable can be used.

```
// bankaccount.cc
#include "bankaccount.hh"

BankAccount::BankAccount() {
    balance = 0;
}
```

Note: not every instance variable needs to be explicitly initialized. For instance, built-in data structures (e.g., **vector**, **map**) are automatically initialized to be empty when they are declared.

Constructor

A class can have more than one constructor, and constructors can also take in parameters. For example, let’s say we want to optionally let the client specify the initial account balance:

```
// client code
BankAccount ba1;      // balance initialized to 0
BankAccount ba2(50);  // balance initialized to 50
```

Constructor

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here
    // constructor
    BankAccount();
    BankAccount(double initialAmount);

private:
    // private internal-only functionality/data goes here
    double balance;
};
```

Constructor

```
// bankaccount.cc
#include "bankaccount.hh"

BankAccount::BankAccount() {
    balance = 0;
}

BankAccount::BankAccount(double initialAmount) {
    balance = initialAmount;
}
```

Constructor

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible
functionality/data goes here
    BankAccount(double initialAmount);

private:
    // private internal-only
functionality/data goes here
    double balance = 0;
};
```

In this case, could we also initialize the balance directly in the header file, and then not need to have one of the constructors?

Constructor

```cpp
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here
    BankAccount(double initialAmount);

private:
    // private internal-only functionality/data goes here
    double balance = 0;
};
```

In this case, could we also initialize the balance directly in the header file, and then not need to have one of the constructors? **Yes**.

However, it’s sometimes more common to initialize in the constructor; sometimes, we cannot initialize in the header (e.g., if its initialization depends on a constructor parameter).

Overview

- What are classes?
- Defining a Class
- Instance Variables
- Constructor
- **Methods**
- Destructor

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Methods

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here
    BankAccount();
    BankAccount(double initialAmount);

    // public methods
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();

private:
    // private internal-only functionality/data goes here
    double balance;
};
```

Methods

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here
    BankAccount();
    BankAccount(double initialAmount);

    // public methods
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();

private:
    // private internal-only functionality/data goes here
    void helperFn(); // private methods for internal-only use (e.g. helpers)
    ...
```

Methods

```
// bankaccount.cc
#include "bankaccount.hh"

BankAccount::BankAccount() {
    balance = 0;
}

BankAccount::BankAccount(double initialAmount) {
    balance = initialAmount;
}

void BankAccount::deposit(double amount) {
    if (amount > 0) {
        balance += amount;
    }
}

...
```

Methods

```
// bankaccount.cc
#include "bankaccount.hh"

BankAccount::BankAccount() {
    balance = 0;
}

BankAccount::BankAccount(double initialAmount) {
    balance = initialAmount;
}

void BankAccount::deposit(double amount) {
    if (amount > 0) {
        balance += amount;
    }
}

...
```

`BankAccount::` means the method is within the BankAccount class, so it can access all members (e.g. instance variables).

Methods

```
// bankaccount.cc
#include "bankaccount.hh"
...

void BankAccount::withdraw(double amount) {
    if (amount <= balance) {
        balance -= amount;
    }
}

double BankAccount::getBalance() {
    return balance;
}
```

Methods

```
// bankaccount.cc
#include "bankaccount.hh"

...

void myHelper() {
    if (balance > 0) {  // error: not a BankAccount method, can’t access balance!
        ...
}
```

If you want to make helper functions that need to access class members (e.g. instance variables), make sure to define them in the **private** section of the header and implement them as `ClassName::method`. If you instead just declare a regular helper function, you cannot access class members!

Methods

```
// bankaccount.hh
#pragma once

class BankAccount {
...
private:
    void myHelper();
...
};
```

```
// bankaccount.cc
#include "bankaccount.hh"


...

void BankAccount::myHelper() {
    if (balance > 0) {  // ok!
        ...
```

this

Methods are “called on” a particular instance and operate on a particular instance. Here, deposit is “called on” **ba1:**

```
BankAccount ba1;
ba1.deposit(2.00);
```

Within methods, the **this** keyword is a pointer to the instance the method is called on:

```
void BankAccount::deposit(double amount) {
    // for above code, "this" would be a pointer to ba1
    cout << this << endl;  // prints address of current object
```

this

Why is the **this** keyword useful? Useful in specific cases such as:

- Needing to explicitly specify that we are referring to an instance variable vs. a local variable:

```
BankAccount::BankAccount(double balance) {
    this->balance = balance;
}
```

(side note – an easy way to avoid the need for this is to just use separate names for the parameter and instance variable)

- If you need to get a pointer to the current object to pass as a parameter or save somewhere.

Overview

- What are classes?
- Defining a Class
- Instance Variables
- Constructor
- Methods
- **Destructor**

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Destructor

A class's **destructor** is called *automatically* when a variable of that type is destroyed. It is destroyed if it goes out of scope or is deleted from the heap. The destructor should do any internal cleanup required before the object goes away.

```
// client code
if (...) {
    BankAccount ba(50);
    ...
} // triggers destructor for ba
```

Destructor

A class's **destructor** is called *automatically* when a variable of that type is destroyed. It is destroyed if it goes out of scope or is deleted from the heap. The destructor should do any internal cleanup required before the object goes away.

```
// client code
BankAccount *ba = new BankAccount();
...
delete ba;  // triggers destructor for ba
```

You do not need to define a destructor in your class if you don't need one (if you don't, C++ will just pretend you defined a destructor that is empty).

Destructor

Destructors are commonly used to free any heap-allocated instance variables (if any). You may also have other necessary logic that needs to run when an object goes away.

Destructor

```
// bankaccount.hh
#pragma once

class BankAccount {
public:
    // client-accessible functionality/data goes here
    BankAccount();
    BankAccount(double initialAmount);

    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();

    // destructor
    ~BankAccount();
private:
    // private internal-only functionality/data goes here
    double balance;
```

Destructor

```
// bankaccount.cc
#include "bankaccount.hh"
...

BankAccount::~BankAccount() {
   // code here runs before the instance goes away
}
```

Other Notes

- **"using namespace std"** at the top of cc files, and **std**:: in front of types in header files:
  - A namespace is like a named grouping and allows us to have multiple things in our program with the same. Common built-in C++ types like **vector, map, string,** etc. are in the "std" namespace.
  - By default, we need to include std:: in front of these types to tell C++ where to find them.
  - However, we commonly avoid this in .cc files by putting **using namespace std** at the top of the file. (Tells C++: "assume I'm referring to the std namespace if I don't specify and you don't know which namespace something is in")
  - You usually shouldn't put "using namespace std" at the top of header (.h/.hh) files because this will cause that namespace to be included in any file that imports it, which they may not want. For this reason, you'll commonly see std:: in header files where we need to refer to std types.

Other Notes

- **Classes vs structs:** similar, but structs default to public access for all members. Classes default to private members, which encourages the idea of *abstraction*: only exposing functionality and data that is important for the client to see.
- Anything in the **private** section is accessible only within that class. You can define various components there: instance variables, private methods, struct definitions, etc.

Demo

bankaccount.hh, bankaccount.cc, client.cc

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Recap

- What are classes?
- Defining a Class
- Instance Variables
- Constructor
- Methods
- Destructor

```
cp -r /afs/ir/class/cs111/lecture-code/classes .
```

Pointers and Memory Review

Additional Resources:

Sean Szumlanski's awesome CS106B lecture notes for C++ (pointers/arrays and heap)

CS107 prior quarter course website for slides for C (strings, pointers, heap)

Overview

- Pointers overview
- ”Address of” vs. ”pass by reference”
- C arrays vs. pointers
- C Strings
- Heap Allocation

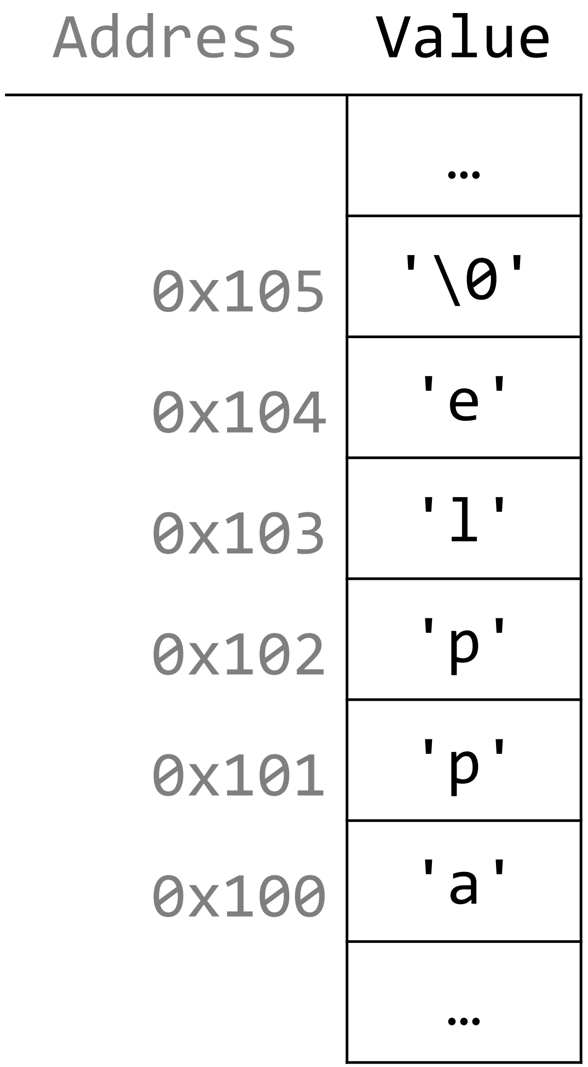
Overview

- **Pointers overview**
- ”Address of” vs. ”pass by reference”
- C arrays vs. pointers
- C Strings
- Heap Allocation

Pointers and Memory

A *pointer* is a variable that stores a memory address.

- Memory is a big array of bytes, and each byte has a unique numeric index that is commonly written in hexadecimal. A pointer stores one of these “indexes”.
- Pointers are also essential for allocating memory on the heap.



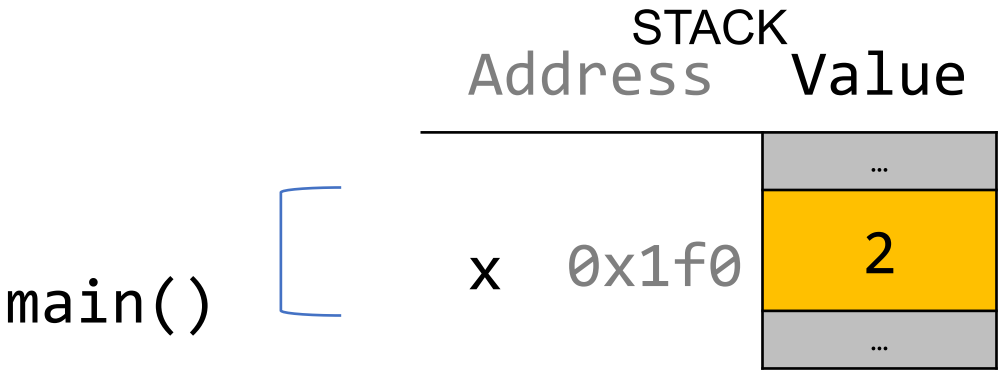
Address	Value
	…
0x105	'\0'
0x104	'e'
0x103	'l'
0x102	'p'
0x101	'p'
0x100	'a'
	…

Pass By Value

When you pass a value as a parameter, C/C++ passes a copy of that value.

```
void myFunc(int val) {
	val = 3;
}

int main(int argc, char *argv[]) {
	int x = 2;
	myFunc(x);
	printf("%d", x);    // 2!
	...
}
```



STACK

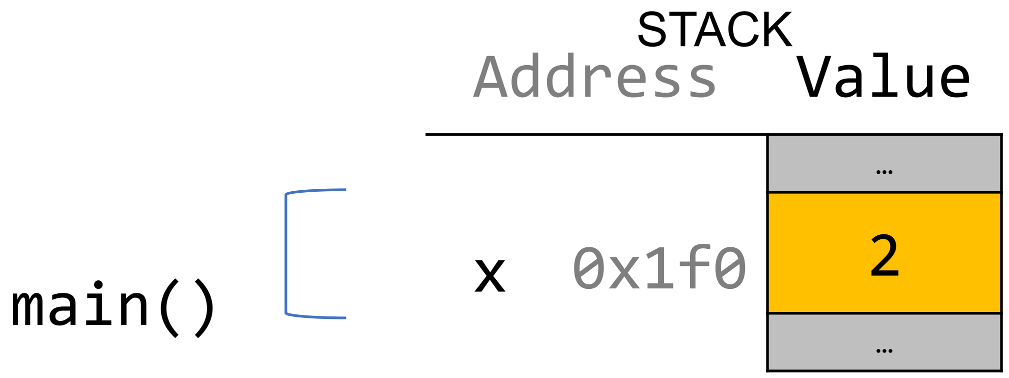
	Address	Value
		…
x	0x1f0	2
		…

Pass By Value

When you pass a value as a parameter, C/C++ passes a copy of that value.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);    // 2!
     ...
}
```

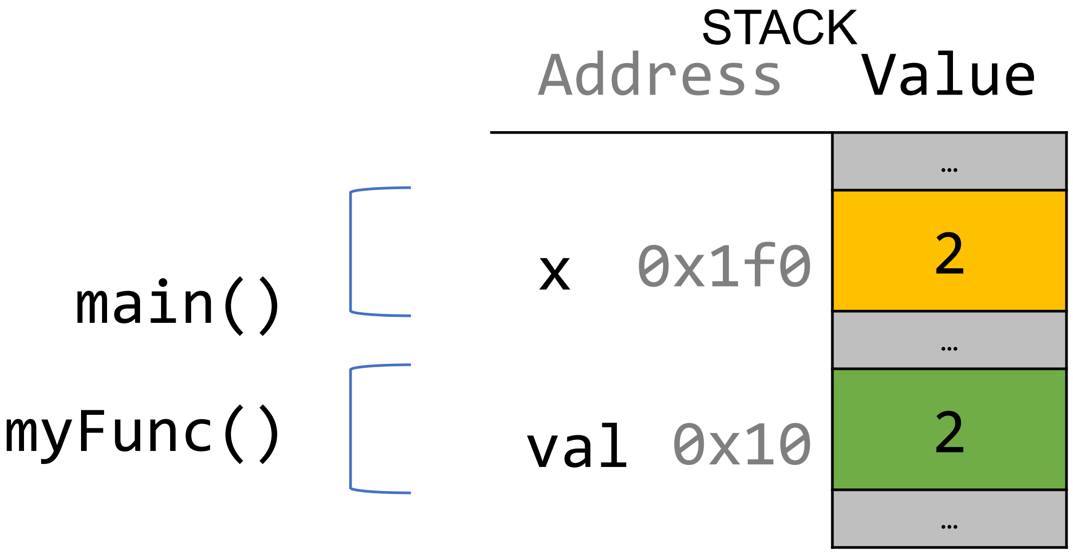


Pass By Value

When you pass a value as a parameter, C/C++ passes a copy of that value.

```
void myFunc(int val) {
      val = 3;
}

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(x);
      printf("%d", x);    // 2!
      ...
}
```

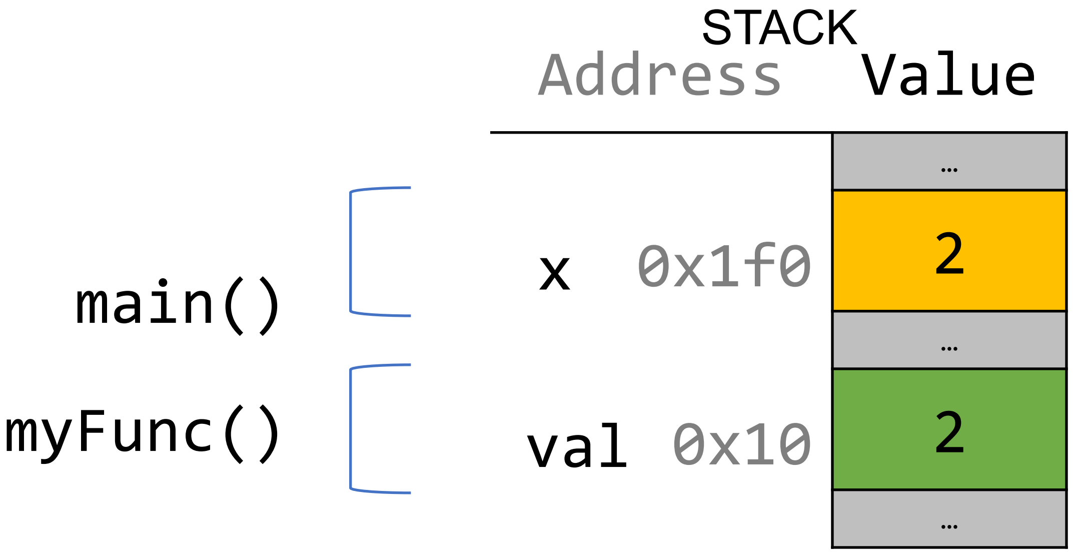


Pass By Value

When you pass a value as a parameter, C/C++ passes a copy of that value.

```
void myFunc(int val) {
    val = 3;
}

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(x);
    printf("%d", x);    // 2!
    ...
}
```

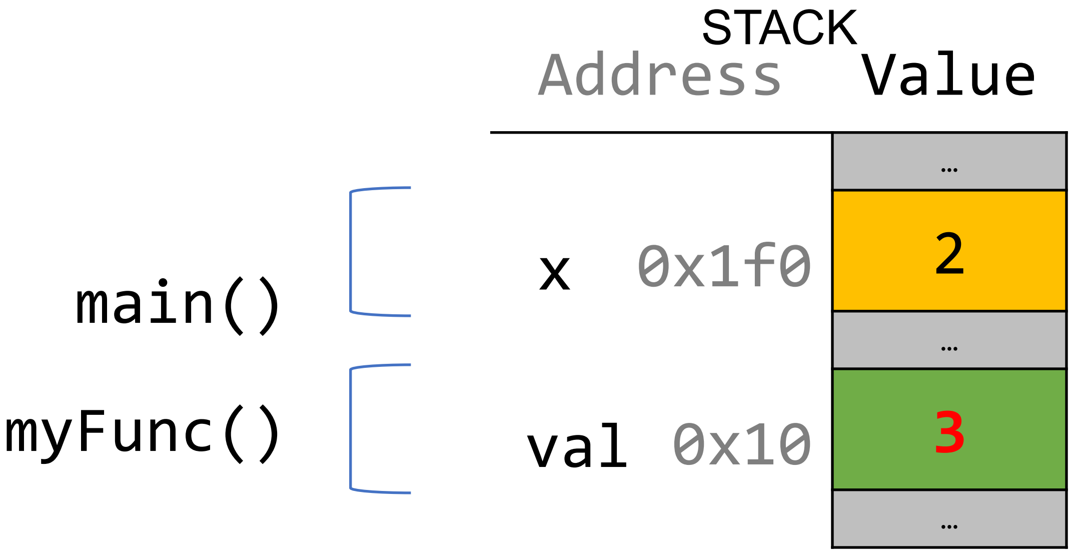


Pass By Value

When you pass a value as a parameter, C/C++ passes a copy of that value.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);     // 2!
     ...
}
```

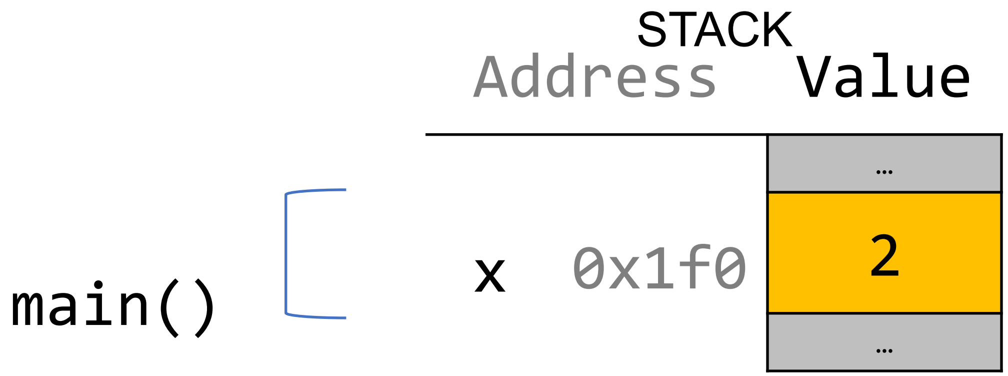


Pass By Value

When you pass a value as a parameter, C/C++ passes a copy of that value.

```
void myFunc(int val) {
     val = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(x);
     printf("%d", x);    // 2!
     ...
}
```



Pointers

Pointers allow us to pass around the *location* of data so that the original data can be modified in other functions.

Example: I want to write a function *myFunc* that can change the value of an existing integer to be 3.

```
int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(???);
    printf("%d", x);   // want to print 3
    ...
}
```

Pointers

```
int x = 2;

// Make a pointer that stores the address of x.
// (& means "address of")
int *xPtr = &x;

// Dereference the pointer to go to that address.
// (* means "dereference")
printf("%d", *xPtr);    // prints 2
```

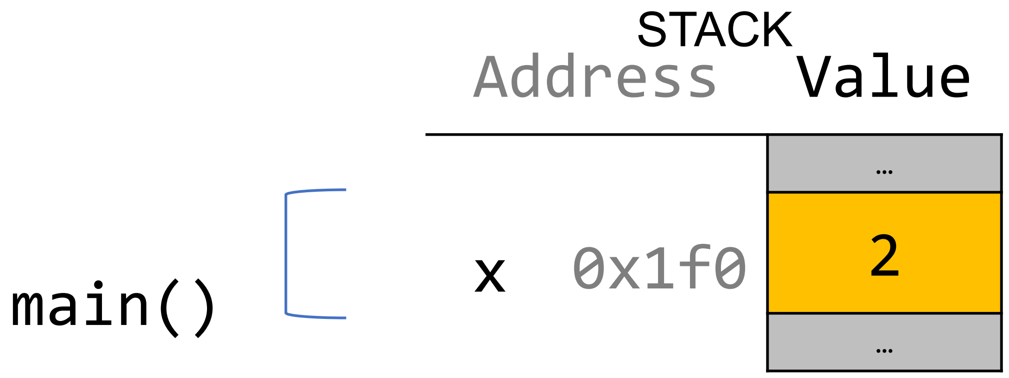
* If **declaration**: “pointer”
ex: `int *` is "pointer to an int”
If **operation**: "dereference/the value at address”
ex: *num is "the value at address num"

Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```



STACK

Address Value

…

main() x 0x1f0 2

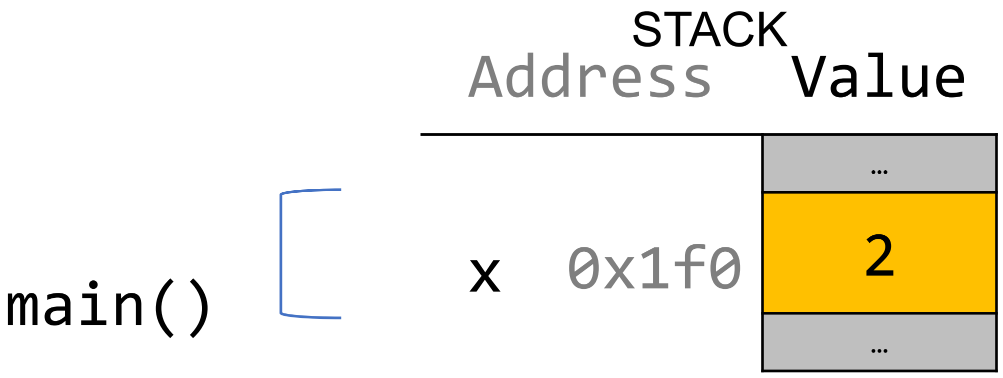
…

Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```

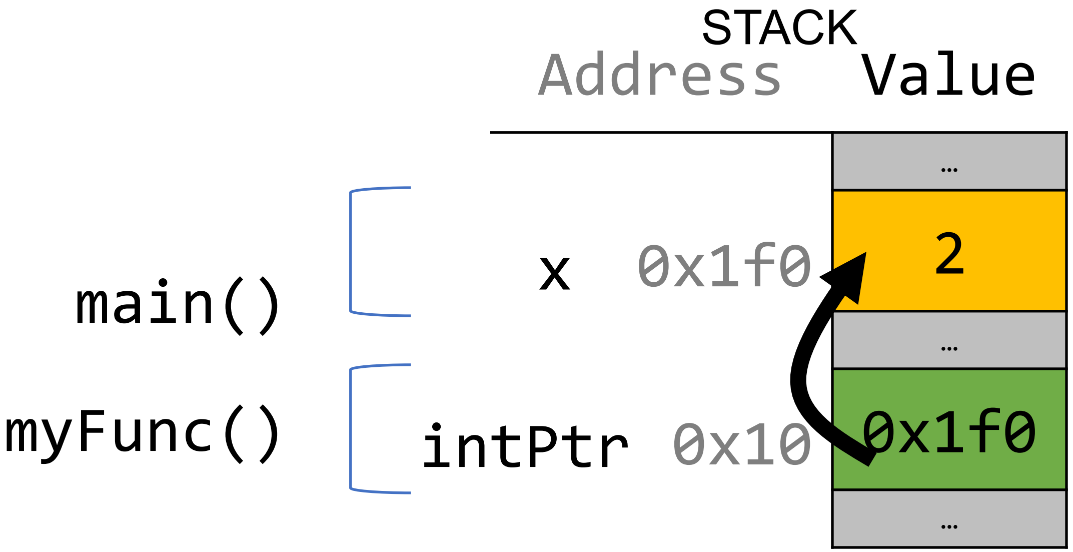


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```

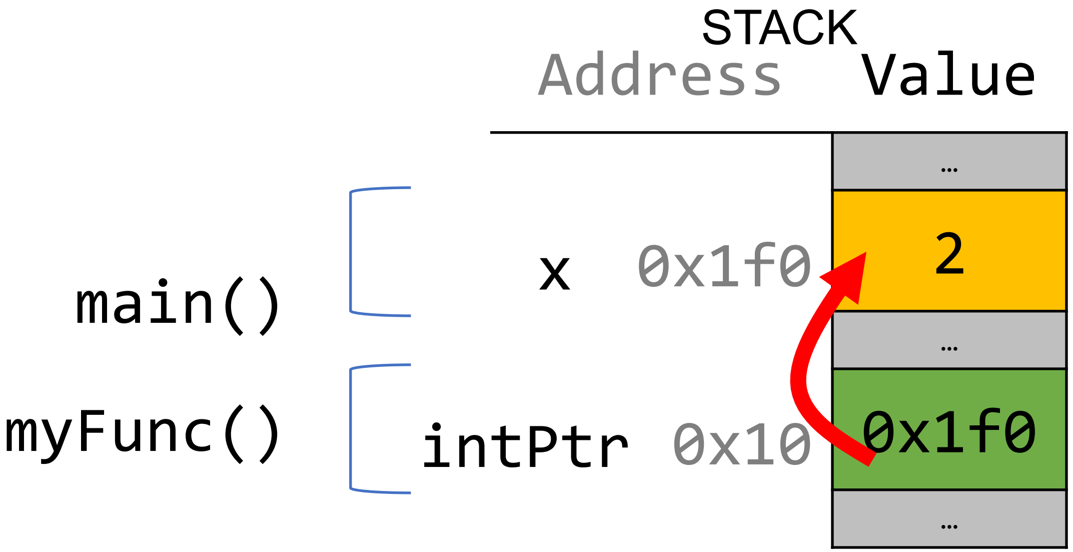


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

int main(int argc, char *argv[]) {
      int x = 2;
      myFunc(&x);
      printf("%d", x);    // 3!
      ...
}
```

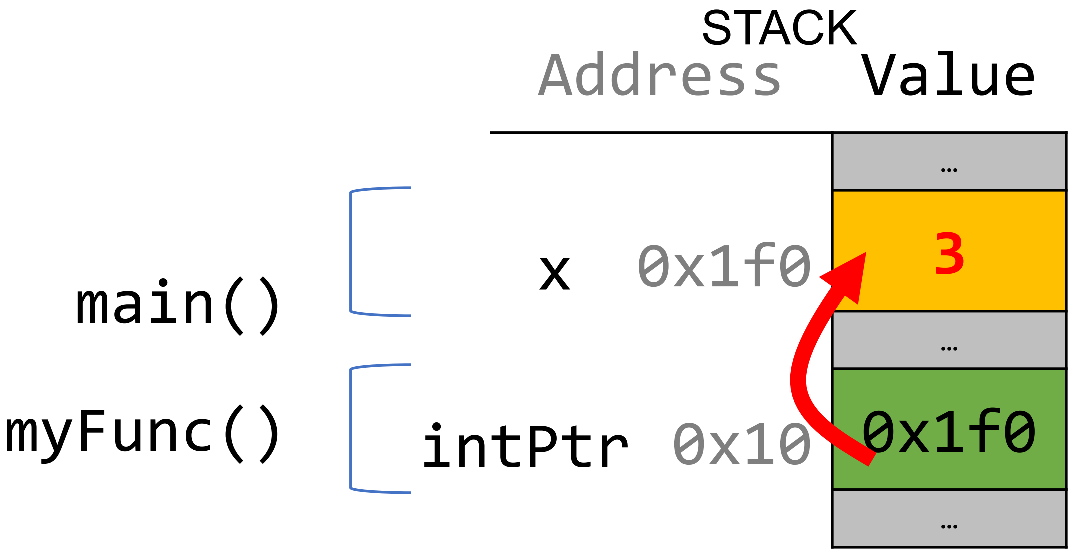


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

int main(int argc, char *argv[]) {
    int x = 2;
    myFunc(&x);
    printf("%d", x);    // 3!
    ...
}
```

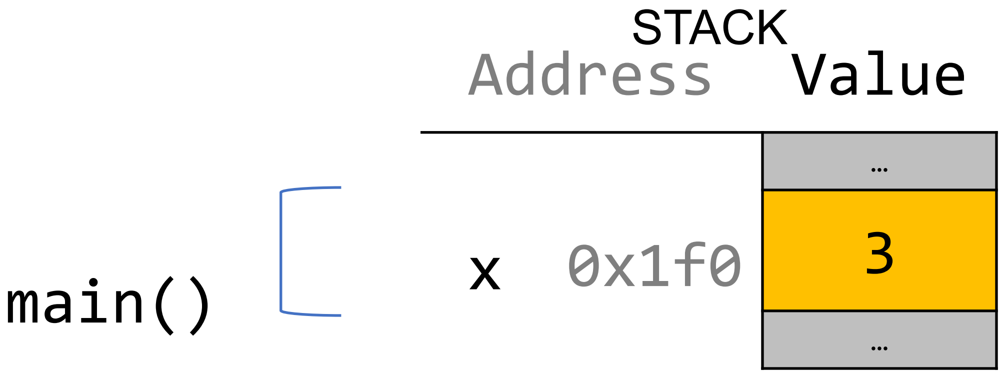


Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

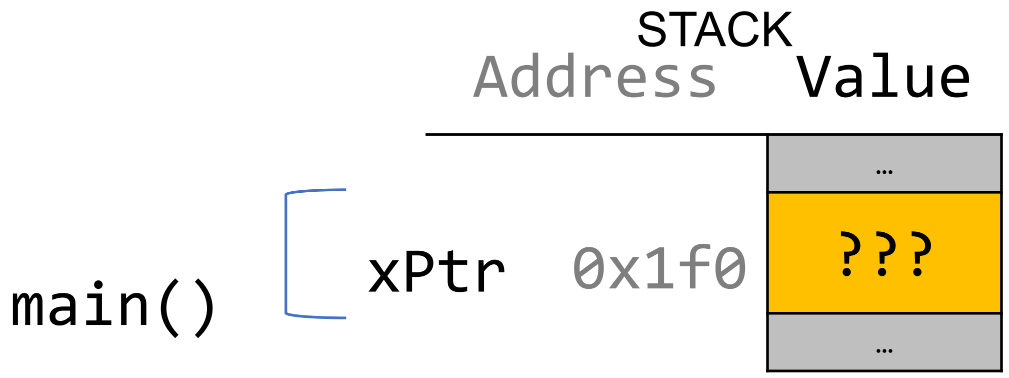
```
void myFunc(int *intPtr) {
     *intPtr = 3;
}

int main(int argc, char *argv[]) {
     int x = 2;
     myFunc(&x);
     printf("%d", x);    // 3!
     ...
}
```



Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.



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

int main(int argc, char *argv[]) {
     int *xPtr;
     myFunc(xPtr);
     ...
}
```

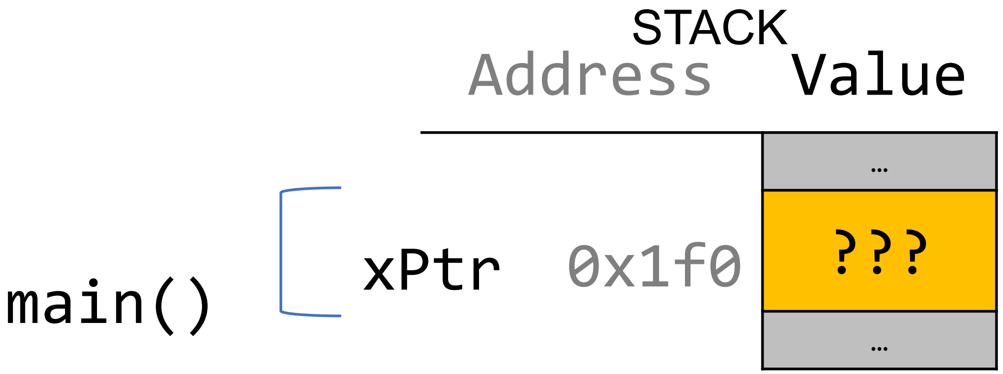
Would this also work?

Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

int main(int argc, char *argv[]) {
     int *xPtr;
     myFunc(xPtr);
     ...
}
```



STACK

		Address	Value
			…
main()	xPtr	0x1f0	???
			…

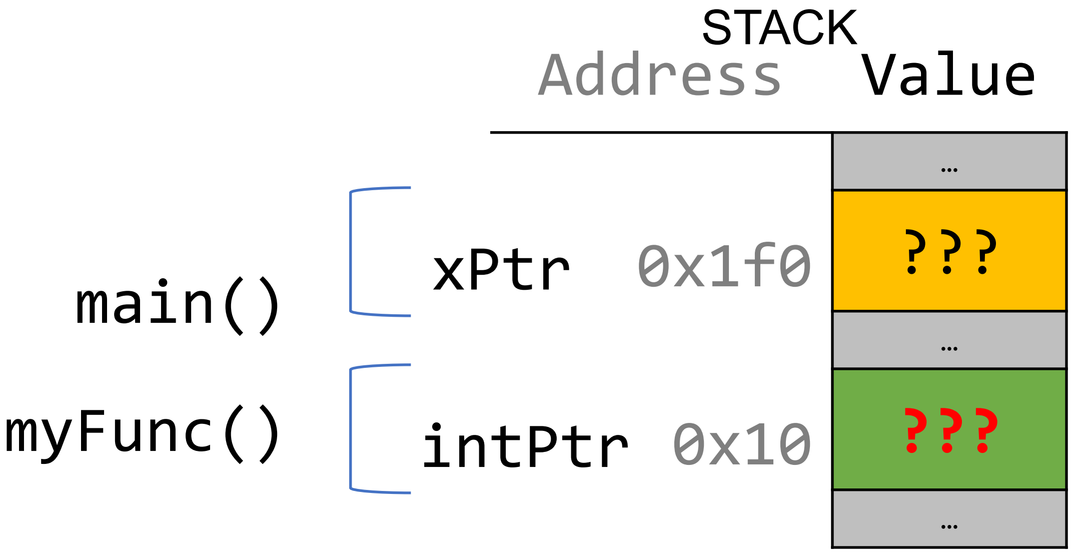
Would this also work? The types match, and this would compile, but this **wouldn't work**. The reason is we are not making space for an int; we are making space for a pointer, which is not initialized. **myFunc** will then try to go to the memory address stored in the pointer, even though the pointer is not referring to valid memory!

Pointers

A pointer lets us pass **where** a particular instance of data is, so it can be modified.

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

int main(int argc, char *argv[]) {
     int *x;
     myFunc(x);
     printf("%d", *x);
     ...
}
```



STACK

Address Value

...

main() xPtr 0x1f0 ???

...

myFunc() intPtr 0x10 ???

...

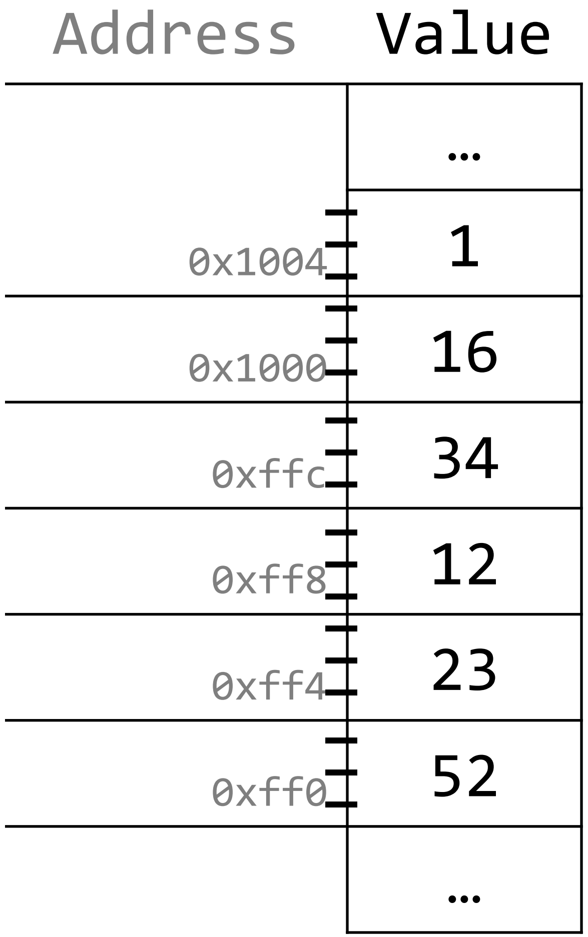
Would this also work? The types match, and this would compile, but this **wouldn’t work**. The reason is we are not making space for an int; we are making space for a pointer, which is not initialized. **myFunc** will then try to go to the memory address stored in the pointer, even though the pointer is not referring to valid memory!

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

printf("%d", *nums);    // 52
printf("%d", *nums1);   // 23
printf("%d", *nums3);   // 34
```



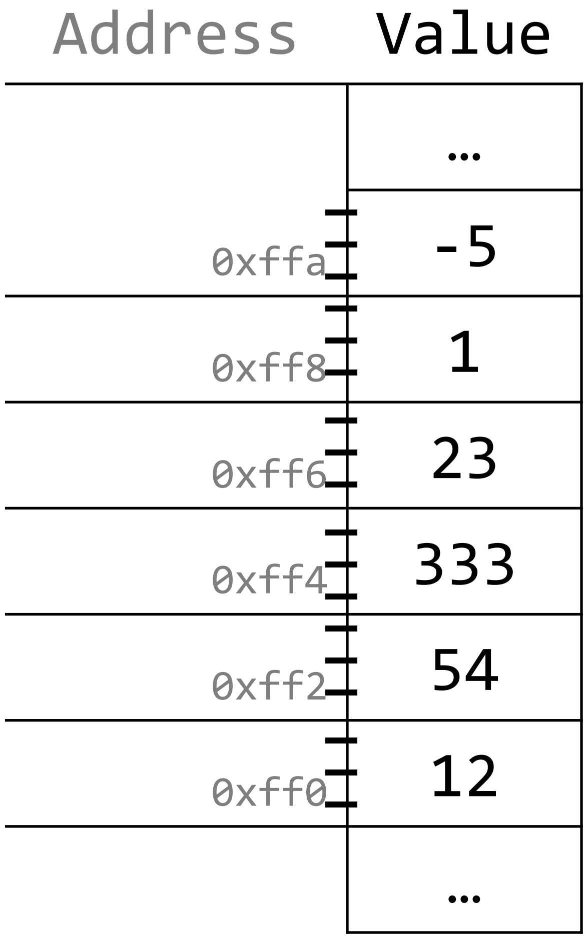
Address	Value
	…
0x1004	1
0x1000	16
0xffc	34
0xff8	12
0xff4	23
0xff0	52
	…

Pointer Arithmetic

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

```
// nums points to a short array
short *nums = …              // e.g. 0xff0
short *nums1 = nums + 1; // e.g. 0xff2
short *nums3 = nums + 3; // e.g. 0xff6

printf("%h", *nums);    // 12
printf("%h", *nums1);   // 54
printf("%h", *nums3);   // 23
```

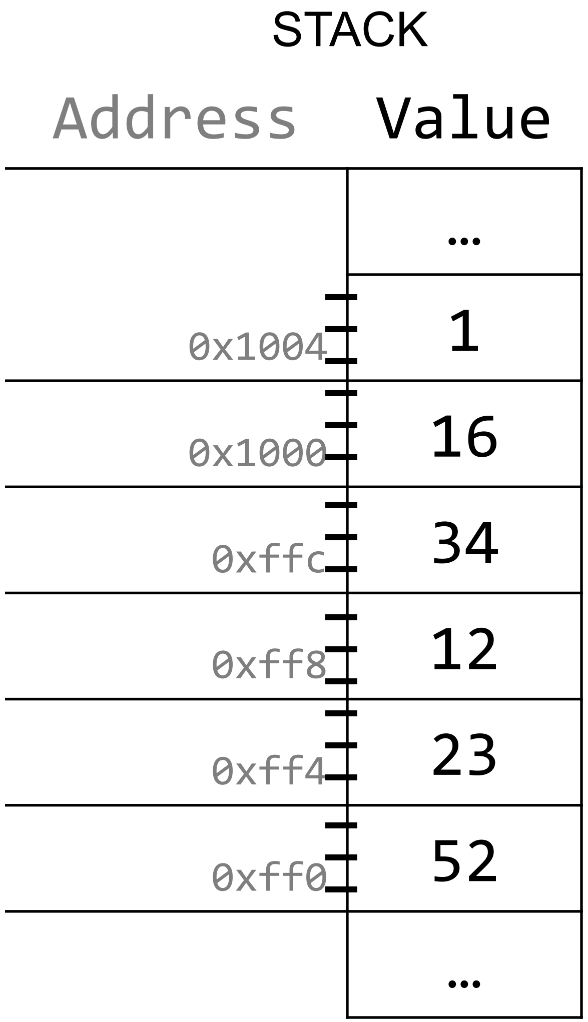


Address	Value
	…
0xffa	-5
0xff8	1
0xff6	23
0xff4	333
0xff2	54
0xff0	12
	…

Pointer Arithmetic

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
```



STACK

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

Structs

A *struct* is a way to define a new variable type that is a group of other variables.

```
typedef struct {
    int month;
    int day;
} date;
…

date today;                       // construct structure instances
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};    // shorter initializer syntax
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
void advance_day(date *d) {
     d->day++;       // equivalent to (*d).day++;
}

int main(int argc, char *argv[]) {
     date my_date = {1, 28};
     advance_day(&my_date);
     printf("%d", my_date.day); // 29
     return 0;
}
```

Overview

- Pointers overview
- **”Address of” vs. ”pass by reference”**
- C arrays vs. pointers
- C Strings
- Heap Allocation

“Address of” vs. Pass by Reference

- In both C and C++, the ”&” operator is the “address of” operator; it gets the address of a variable.
- In C++, ”&” also has another (!) common meaning, which is confusing; when used in a function signature, it means you are passing a parameter by reference.

```
void myFunc(int& num) {
    num = 3;  // DOES change the original value in the caller!
}

int main() {
    int x = 2;
    myFunc(x);
    // now x is 3!
    ...
```

“Address of” vs. Pass by Reference

Pass by reference is like an “automatically-dereferenced pointer”; it’s essentially passing a pointer, but automatically dereferencing it before use.

```
void myFunc(int& num) {
    num = 3;  // DOES change the original value in the caller!
}

int main() {
    int x = 2;
    myFunc(x);
    // now x is 3!
    ...
```

“Address of” vs. Pass by Reference

When should we use pointers vs. pass by reference?

- C doesn’t have pass by reference, so we must use pointers in C programs to achieve this kind of behavior.
- In C++, pass by reference is easier to work with than pointers, and can replace some of the uses of pointers in our programs. But we will still need pointers in other places, such as heap allocation.

Overview

- Pointers overview
- ”Address of” vs. ”pass by reference”
- **Arrays vs. pointers**
- C Strings
- Heap Allocation

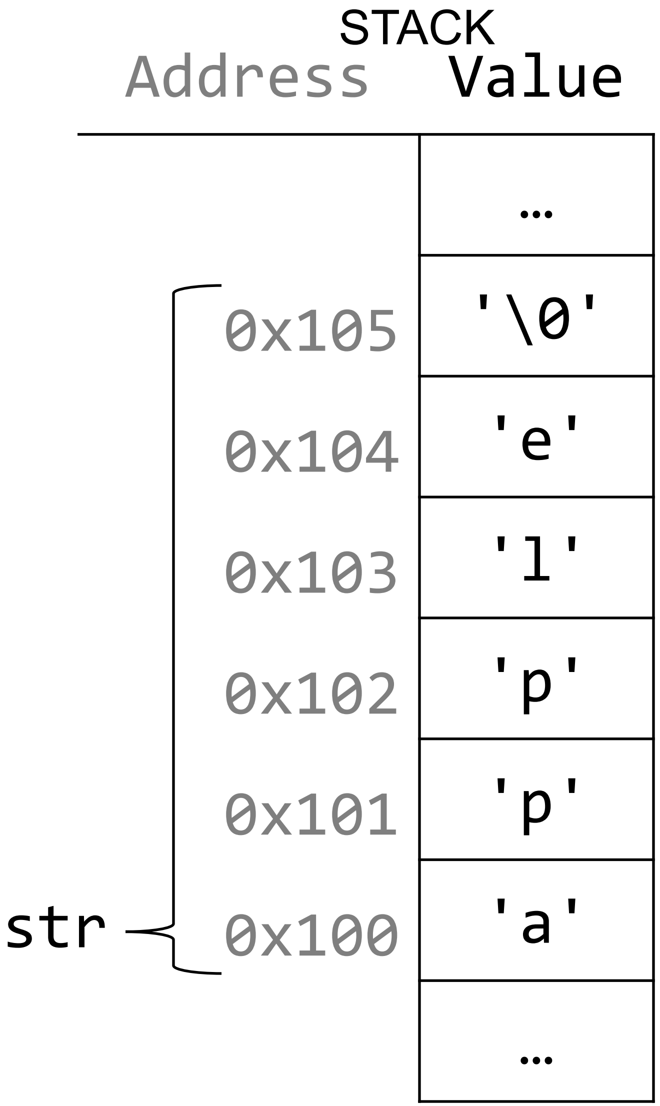
Arrays

When you declare a stack array, contiguous memory is allocated to store the contents of the entire array.

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

The array variable (e.g. **str**) is not a pointer; it refers to the entire array contents. In fact, **sizeof** returns the size of the entire array!

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



STACK

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

Arrays

An array variable refers to an entire block of memory. You 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!
```

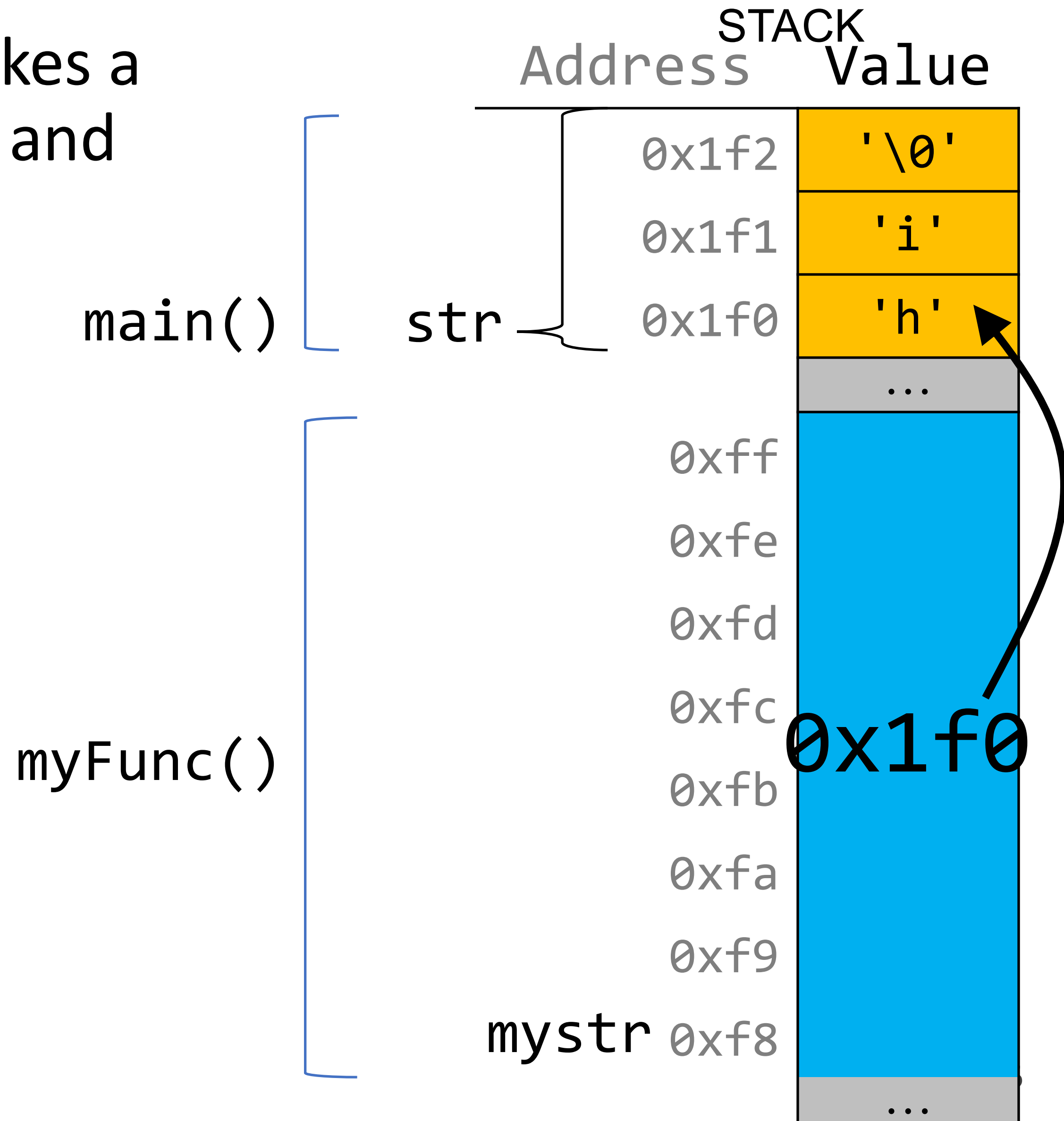
A stack array's size cannot be changed once you create it; you must create another new array instead. (heap-allocated arrays can be resized using **realloc**).

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) {
     ...
}

int main(int argc, char *argv[]) {
     char str[3];
     strcpy(str, "hi");
     myFunc(str);
     ...
}
```

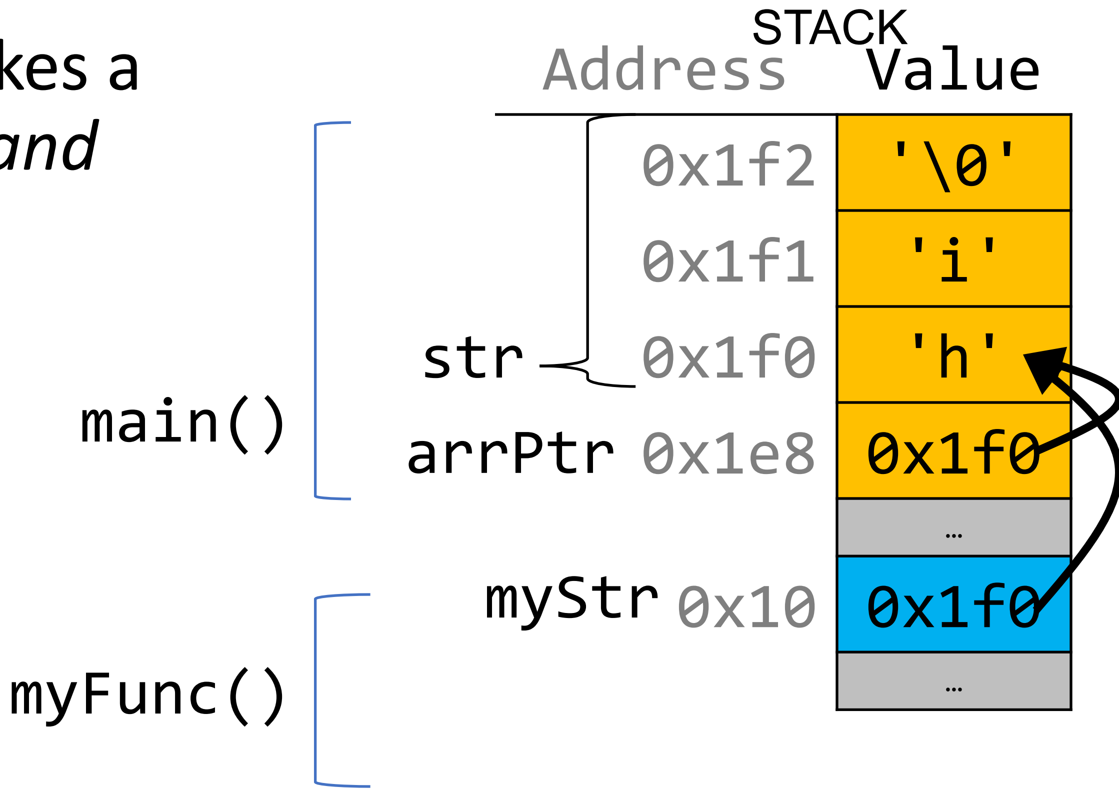


Arrays as Parameters

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) {
     …
}

int main(int argc, char *argv[]) {
     char str[3];
     strcpy(str, "hi");
     // equivalent
     char *arrPtr = str;
     myFunc(arrPtr);
     ...
```

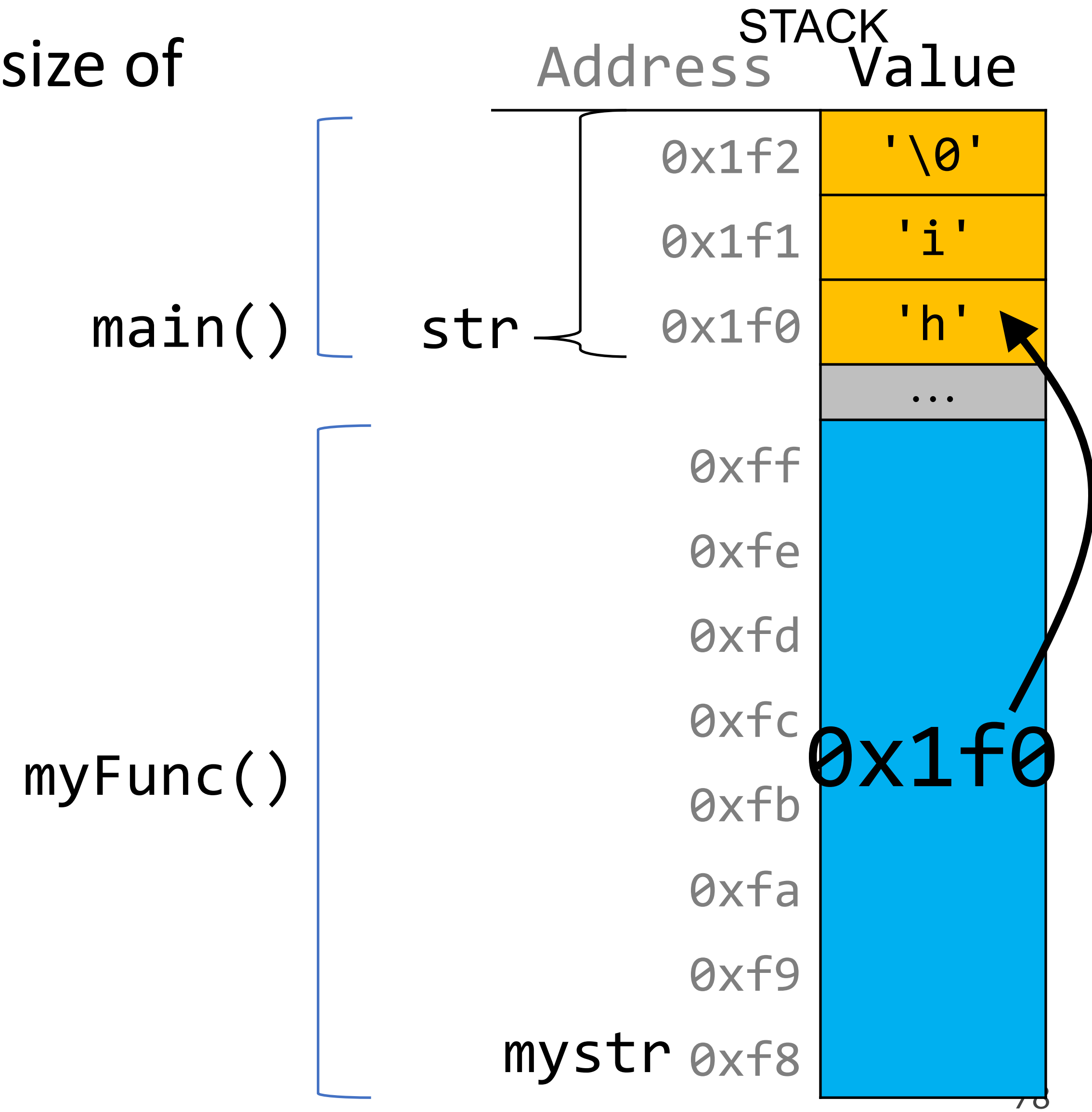


Arrays as Parameters

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

```
void myFunc(char *myStr) {
     int size = sizeof(myStr); // 8
}

int main(int argc, char *argv[]) {
     char str[3];
     strcpy(str, "hi");
     int size = sizeof(str);   // 3
     myFunc(str);
     ...
}
```



Quirks of Arrays

One major quirk of arrays is using **&** directly on an array doesn't do anything (huh?).

```
int nums[4] = {4, 24, 121, -2};
int *ptr = nums;            // 0x100
int *otherPtr = &nums;   // also 0x100!!
```

Why does this happen? An array is not a pointer, but rather represents a chunk of memory storing its elements. When we use &, C says "oh, the address of the array? Sure, here's the starting address of the array". But we can already get that by just setting a pointer equal to the array.

Address	Value
	…
0x10c	-2
0x108	121
0x104	24
0x100	4
	…

& Directly On An Array

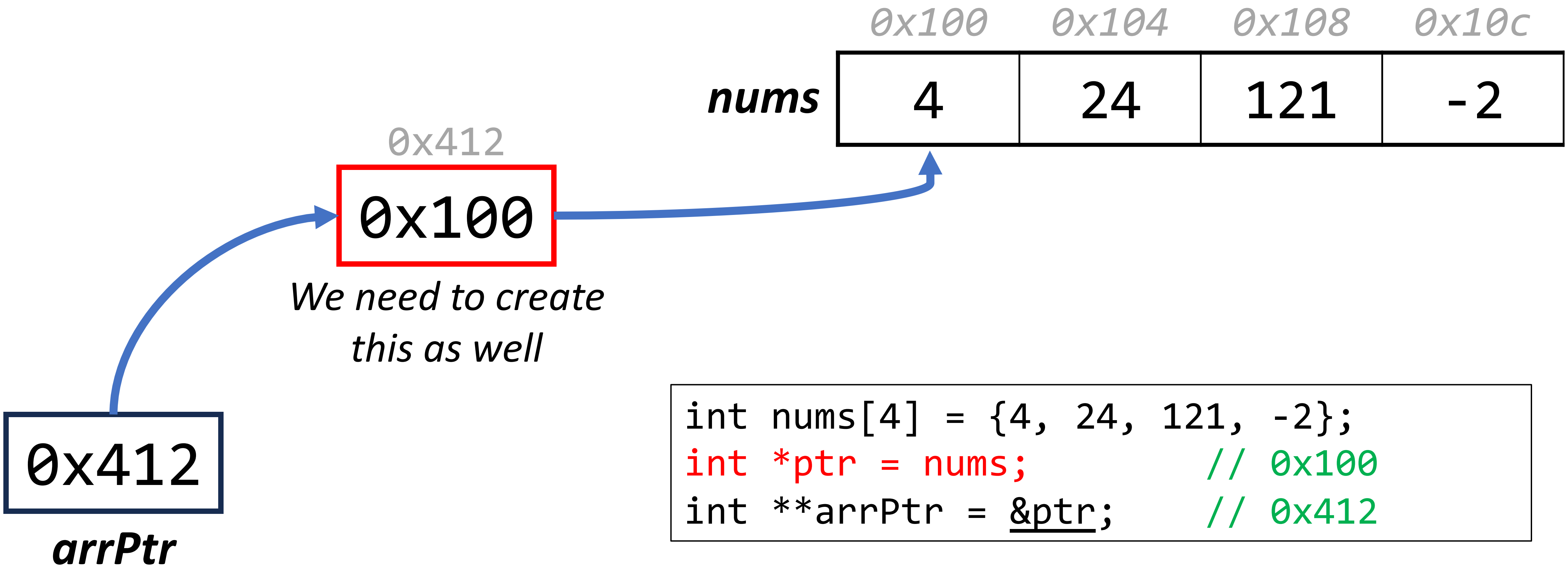
One major quirk of arrays is using **&** directly on an array doesn't do anything.

Example where this causes a problem – let's say we want to get a double pointer to an array (pointer to something that points to the first element):

```
int nums[4] = {4, 24, 121, -2};
// goal
int **arrPtr = ???;
**arrPtr = 15; // should change first elem to 15


// none of these are what we want ☹
int **arrPtr = nums;         // points to first elem
int **arrPtr = &nums;        // also points to first elem!
int **arrPtr = &nums[0];     // also points to first elem!!
```

& Directly On An Array



We need to create this as well

```
int nums[4] = {4, 24, 121, -2};
int *ptr = nums;          // 0x100
int **arrPtr = &ptr;      // 0x412
```

Arrays vs. Pointers Summary

- When you create an array, you are making space for each element in the array.
- When you create a pointer, you are making space for an 8 byte address.
- Arrays ”decay to pointers” when you perform arithmetic or pass as parameters.
- You can set a pointer equal to an array; that pointer will point to the array’s first element
- &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 8

Overview

- Pointers overview
- ”Address of” vs. ”pass by reference”
- C arrays vs. pointers
- **C Strings**
- Heap Allocation

C Strings

C strings are arrays of characters ending with a **null-terminating character** '\0'.

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
value	`'H'`	`'e'`	`'l'`	`'l'`	`'o'`	`','`	`' '`	`'w'`	`'o'`	`'r'`	`'l'`	`'d'`	`'!'`	`'\0'`

String operations such as `strlen` use the null-terminating character to find the end of the string.

Side note: use `strlen` to get the length of a string. Don't use `sizeof`!

Common `string.h` Functions

Function	Description
`strlen(`***str***`)`	returns the # of chars in a C string (before null-terminating character).
`strcmp(`***str1, str2***`)`, `strncmp(`***str1, str2, n***`)`	compares two strings; returns 0 if identical, <0 if ***str1*** comes before ***str2*** in alphabet, >0 if ***str1*** comes after ***str2*** in alphabet. ***strncmp*** stops comparing after at most ***n*** characters.

`strchr(`***str, ch***`)`
`strrchr(`***str, ch***`)`
	character search: returns a pointer to the first occurrence of ***ch*** in ***str***, or ***NULL*** if ***ch*** was not found in ***str***. `strrchr` find the last occurrence.

`strstr(`***haystack, needle***`)`	string search: returns a pointer to the start of the first occurrence of ***needle*** in ***haystack***, or ***NULL*** if ***needle*** was not found in ***haystack***.
`strcpy(`***dst, src***`)`, `strncpy(`***dst, src, n***`)`	copies characters in ***src*** to ***dst***, including null-terminating character. Assumes enough space in ***dst***. Strings must not overlap. **`strncpy`** stops after at most ***n*** chars, and does not add null-terminating char.
`strcat(`***dst, src***`)`, `strncat(`***dst, src, n***`)`	concatenate ***src*** onto the end of ***dst***. **`strncat`** stops concatenating after at most ***n*** characters. Always adds a null-terminating character.
`strspn(`***str, accept***`)`, `strcspn(`***str, reject***`)`	**`strspn`** returns the length of the initial part of ***str*** which contains only characters in ***accept***. **`strcspn`** returns the length of the initial part of ***str*** which does not contain any characters in ***reject***.

Substrings

We can combine pointer arithmetic and copying to make any substrings we'd like.

```
// Want just "ace"
char str1[8];
strcpy(str1, "racecar");

char str2[4];
strncpy(str2, str1 + 1, 3);
str2[3] = '\0';
printf("%s\n", str1);       // racecar
printf("%s\n", str2);       // ace
```

char * vs. char[]

- char * is an 8-byte pointer – it stores an address of a character
- char[] is an array of characters – it stores the actual characters in a string
- When you pass a char[] as a parameter, it is automatically passed as a char * (pointer to its first character)

char * vs. char[]

`char myString[]`

vs

`char *myString`

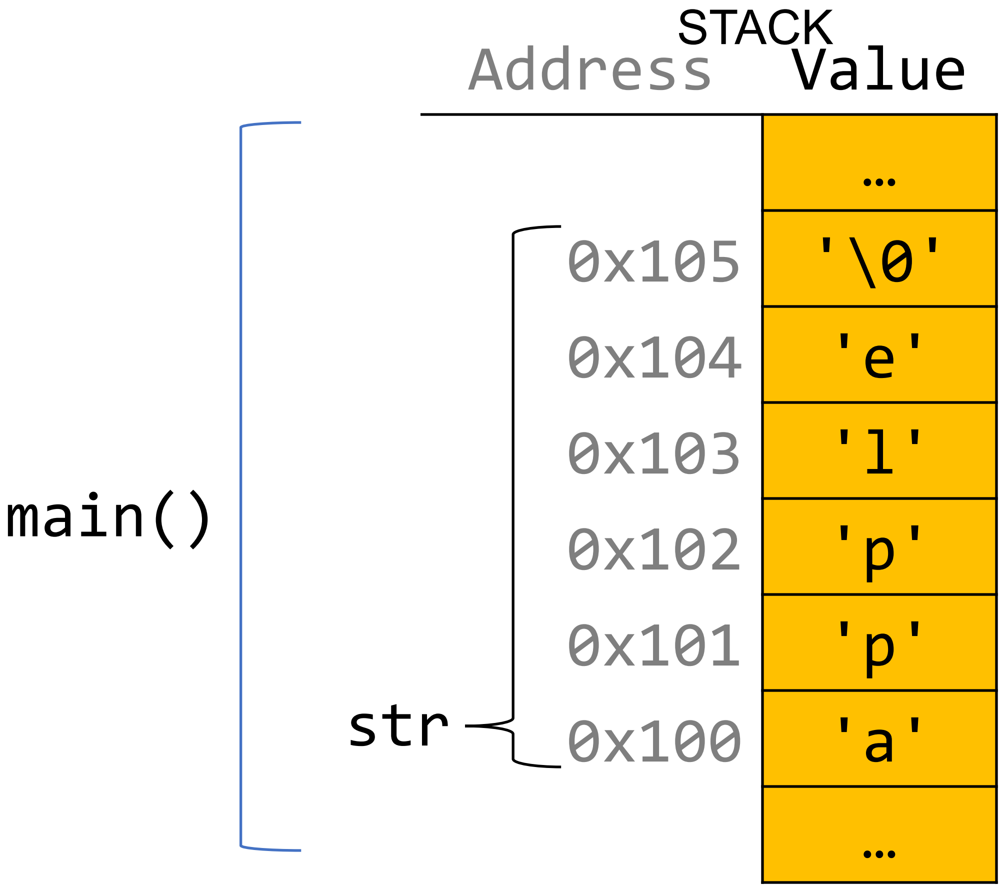
You can create char * pointers to point to any character in an existing string and reassign them since they are just pointer variables. You **cannot** reassign an array.

```
char myString[6];
strcpy(myString, "Hello");
~~myString = "Another string";~~                // not allowed!
---
char *myOtherString = myString;
myOtherString = somethingElse;                  // ok
```

char[]

When we declare an array of characters, contiguous memory is allocated on the stack to store the contents of the entire array.

```
int main(int argc, char *argv[]) {
     char str[6];
     strcpy(str, "apple");
     ...
```



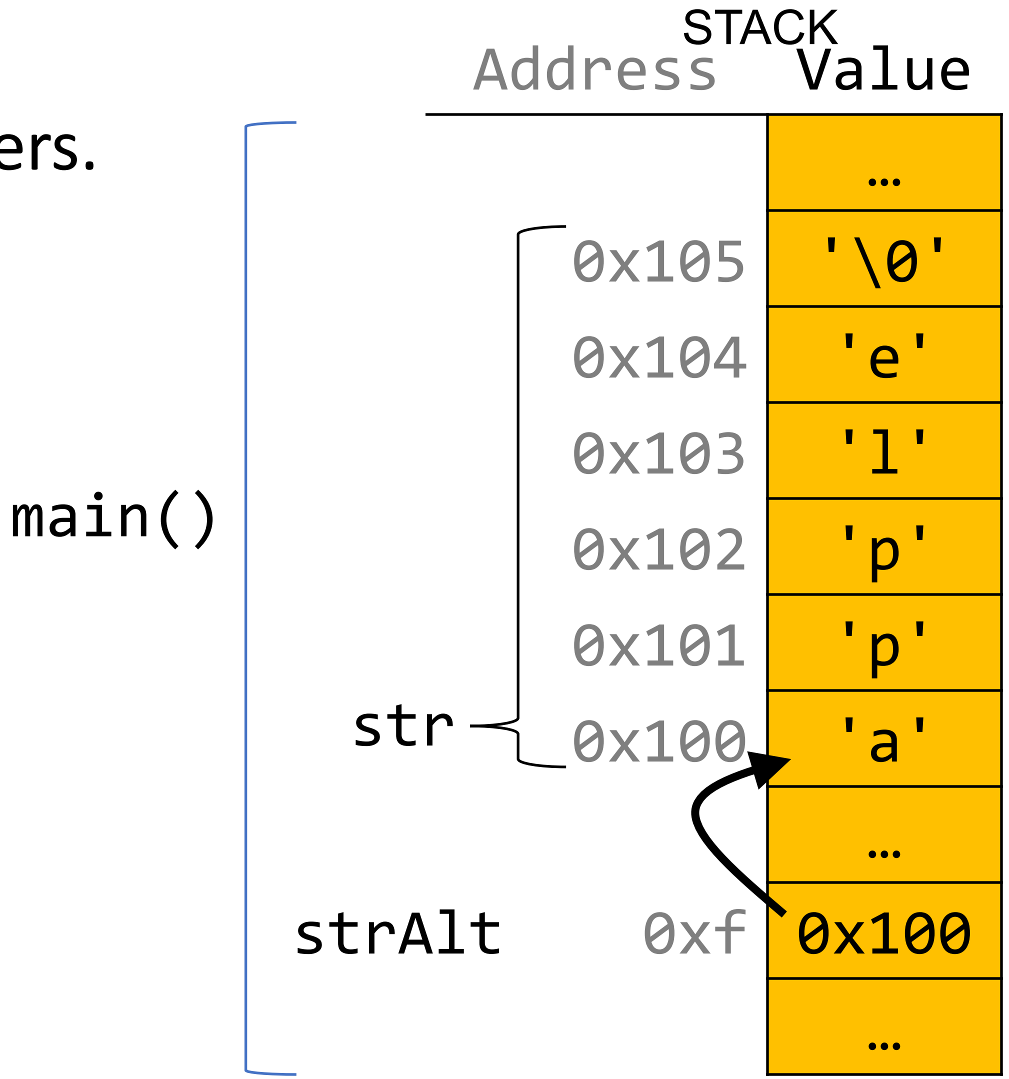
STACK

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

char *

When we declare a **char ***, we allocate space on the stack to store an address, not actual characters. But we can still generally use **char *** the same as **char[]**.

```
int main(int argc, char *argv[]) {
     char str[6];
     strcpy(str, "apple");
     char *strAlt = str;
     ...
```

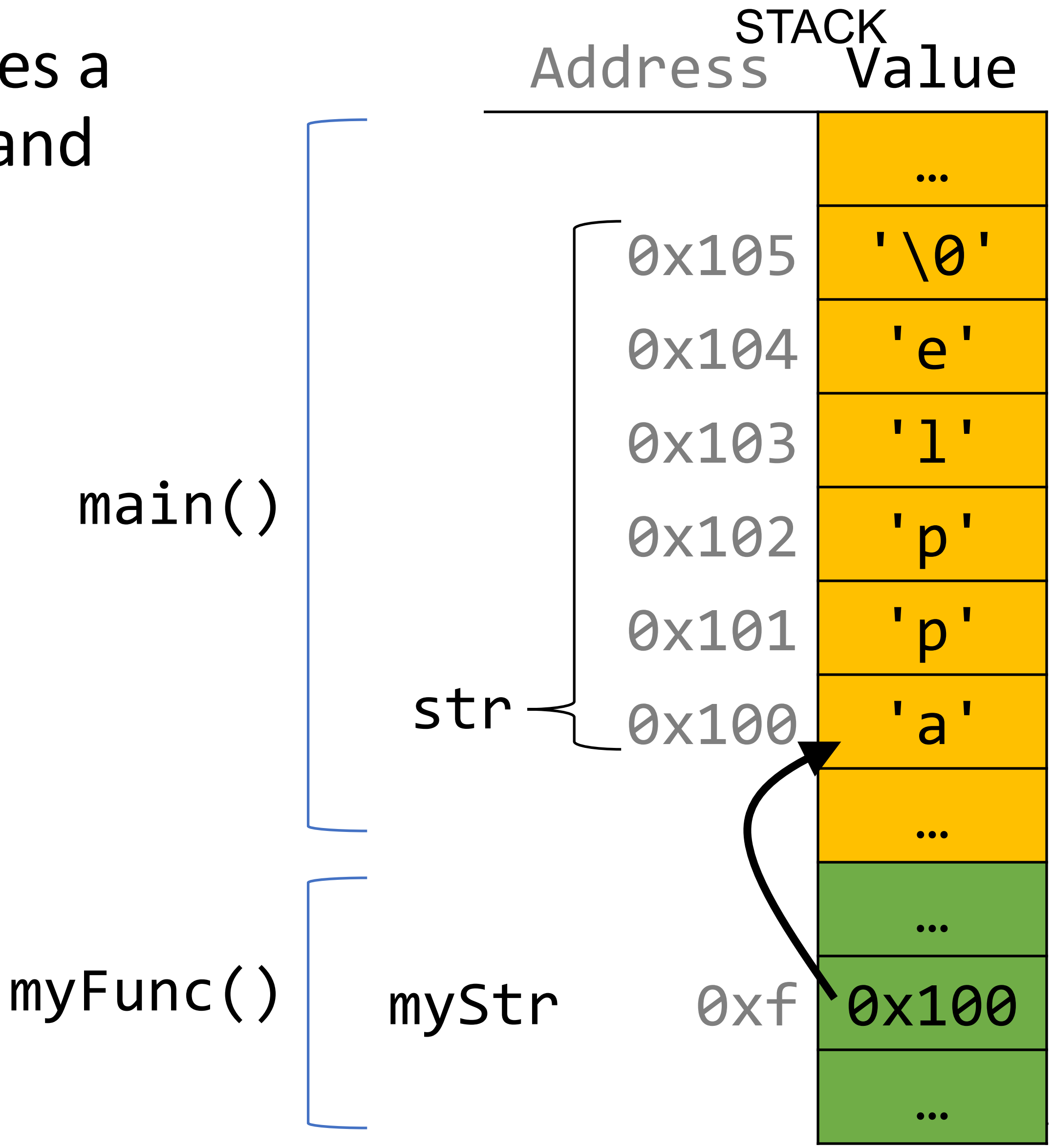


Strings as Parameters

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

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

int main(int argc, char *argv[]) {
	char str[6];
	strcpy(str, "apple");
	myFunc(str);
	...
}
```

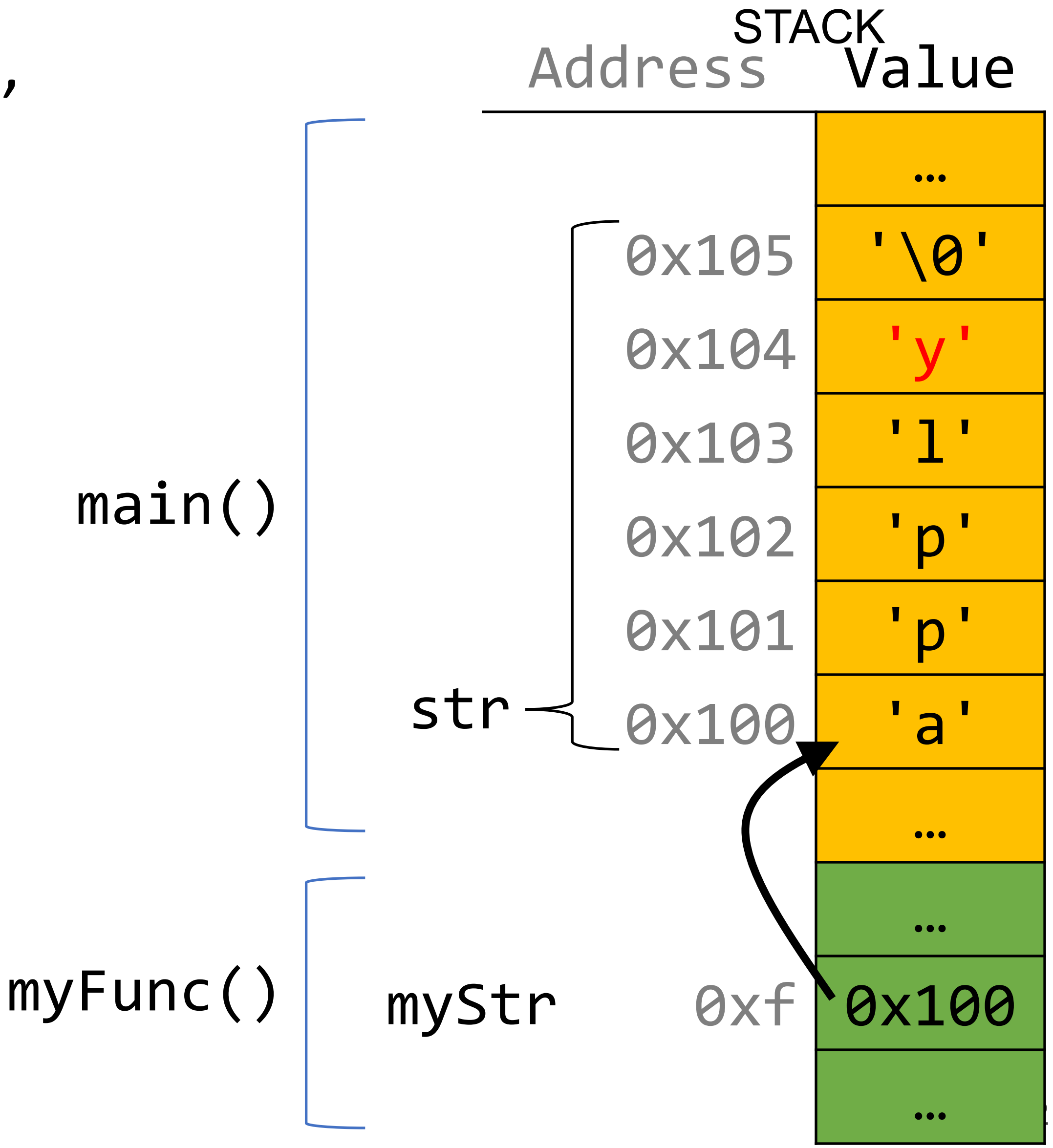


Strings as Parameters

This means if we modify characters in **myFunc**, the changes will persist back in **main**!

```
void myFunc(char *myStr) {
      myStr[4] = 'y';
}

int main(int argc, char *argv[]) {
      char str[6];
      strcpy(str, "apple");
      myFunc(str);
      printf("%s", str);  // apply
      ...
}
```



STACK

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

Key C String Behaviors

1. If we create a string as a **`char[]`**, we can modify its characters because its memory lives in our stack space.
2. We cannot set a **`char[]`** equal to another value, because it is not a pointer; it refers to the block of memory reserved for the original array.
3. If we pass a **`char[]`** as a parameter, set something equal to it, or perform arithmetic with it, it's automatically converted to a **`char *`**.
4. We can set a **`char *`** equal to another value, because it is a reassign-able pointer.
5. Adding an offset to a C string gives us a substring that many places past the first character.
6. If we change characters in a string parameter, these changes will persist outside of the function.

C vs. C++ Strings

C++ has an actual **string** variable type (hooray!) that provides more helpful functionality; for this reason, we want to prefer C++ strings to C strings where we can.

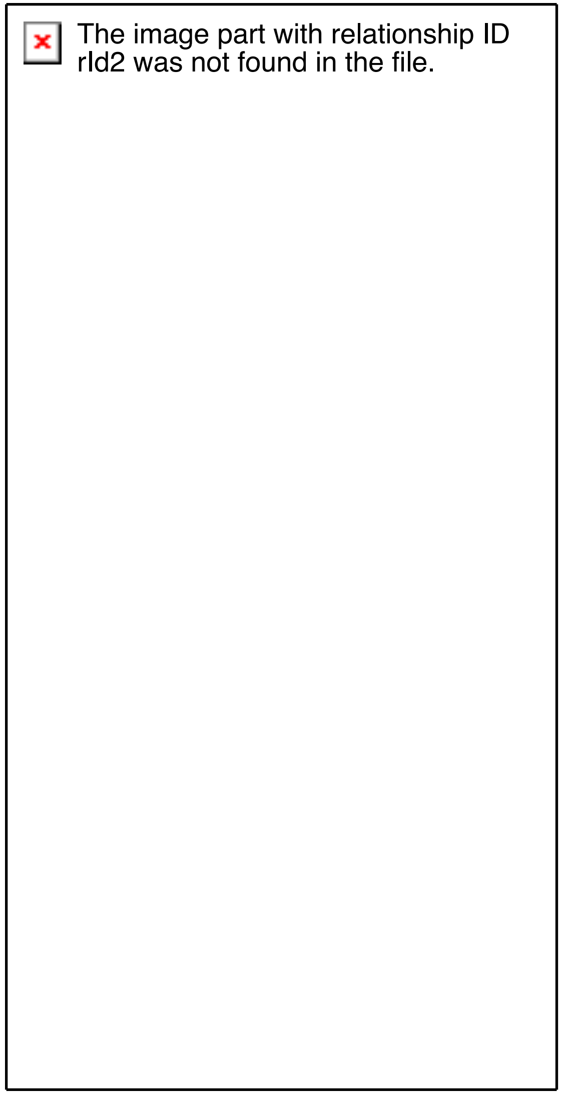
- More C string information: “**man string**” or other resources such as https://cplusplus.com/reference/cstring/
- More C++ string information: resources such as https://cplusplus.com/reference/string/string/

Overview

- Pointers overview
- ”Address of” vs. ”pass by reference”
- C arrays vs. pointers
- C Strings
- **Heap Allocation**

Memory Layout

- The **stack** is the place where all local variables and parameters live for each function. A function's stack "frame" goes away when the function returns.
- The stack grows **downwards** when a new function is called and shrinks **upwards** when the function is finished.
- The **heap** is a part of memory below the stack that you can manage yourself. Unlike the stack, the memory only goes away when you delete it yourself.
- Unlike the stack, the heap grows **upwards** as more memory is allocated.
- The heap is **dynamic memory** – memory that can be allocated, resized, and freed during **program runtime**.



The image part with relationship ID rId2 was not found in the file.

The Heap

- In **C**, we can use **malloc/realloc/free** to manage heap memory.
- In **C++**, we can still use these C functions, but also have **new/delete** to manage heap memory (and prefer these over lower-level functions where possible).

Rule of thumb: we want to default to using the stack unless there is a reason we need heap allocation. Heap allocation is more complex and prone to error, so make sure to only use it where needed! (e.g. we need a resizable array, or we need a variable to not go away, etc.)

Key C Heap Functions

```
void *malloc(size_t size);
```

malloc takes in the number of bytes you want and returns a pointer to the **starting address** of the new memory on the heap. The memory is not initialized!

```
void free(void *ptr);
```

free marks the heap memory pointed to by **ptr** as free, meaning that we will no longer be using it.

```
void *realloc(void *ptr, size_t size);
```

The **realloc** function takes an existing allocation pointer and resizes to a new requested size. It returns a pointer to the start of the larger allocation.

The Heap in C

```
char *strdup(char *s);
```

`strdup` is a convenience function that returns a **null-terminated**, heap-allocated string with the provided text, instead of you having to **`malloc`** and copy in the string yourself.

The Heap in C++

In C++, we don’t need to allocate bytes manually (hooray!). Instead, the **new** operator lets us allocate space for an entire type or array.

```
// Allocating an int on the heap
int *numPtr = malloc(sizeof(int)); // C style, manually allocating 4 bytes
int *numPtr = new int;             // C++ style, less error-prone!

// Allocating an array of 4 ints on the heap
int *nums = malloc(sizeof(int) * 4);  // C style, manually allocating 16 bytes
int *nums = new int[4];               // C++ style, less error-prone!
```

The Heap in C++

In C++, to free a heap allocation, we can use **delete** which, like **free**, deletes the heap memory pointed to by the specified pointer.

```
// Allocating and then deleting an int
int *numPtr = new int;
...
delete numPtr;

// Allocating and then deleting an array (note [] after delete)
int *nums = new int[4];
...
delete[] nums;
```

Memory Leaks

A **memory leak** is when you do not free memory you previously allocated.

- Our program is responsible for cleaning up any memory it allocates but no longer needs.
- If we never free any memory and allocate an extremely large amount, we may run out of memory in the heap!
- However, memory leaks rarely (if ever) cause crashes.
- We recommend not to worry about freeing memory until your program is written. Then, go back and free memory as appropriate.
- Valgrind is a very helpful tool for finding memory leaks! See http://cs107.stanford.edu/resources/valgrind.html for a helpful guide.

Recap

- Pointers overview
- ”Address of” vs. ”pass by reference”
- C arrays vs. pointers
- C Strings
- Heap Allocation