Crash Course C++

Core Concepts Compressed Completely

Why C++

Popular, Fast, Low-Level

- 3rd in Popularity (TIOBE Rating Jan. 2018), only behind Java and C
- 2nd in Average Salary (Indeed.com): $115k
- Extremely Fast (Beat Java/Go/Scala according to Google
- Offers low-level functionality, with object-oriented tools
- Can be used to build almost anything

Evolution of C++

Evolved from C

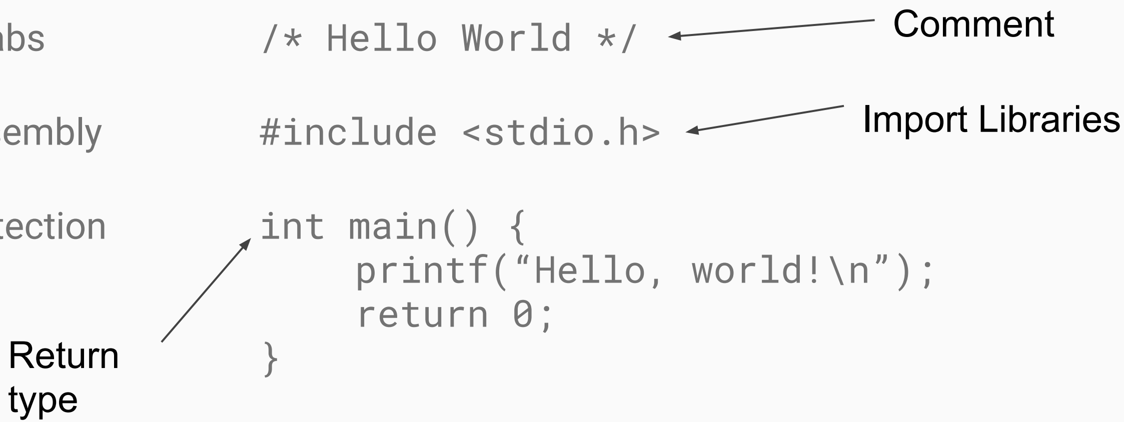
Developed during 1969-1973 in Bell Labs

Acts as a layer of abstraction over assembly

Is extremely flexible, but offers no protection

Very small feature set

Compiled with gcc



```
/* Hello World */

#include <stdio.h>

int main() {
    printf(“Hello, world!\n”);
    return 0;
}
```

Comment

Import Libraries

Return type

C++

Evolved as a superset of C in 1985

Initially called “C with Classes”

Expanded feature set while still maintaining speed

The goal was to let programmers use object-oriented techniques with the power and speed of C

Uses g++

```
/* Hello World */

#include <iostream>

int main() {
    std::cout << “Hello, world”;
    std::cout << std::endl;
    return 0;
}
```

New include style for built in libraries

New functions for I/O

Hello World

Comments

Comments are ignored by compilers

Single line = //

Multi-Line = /* */

The ultimate goal when coding is to write code so magnificent that it doesn’t require comments.

Until then, comment with concise, and meaningful language.

```
/*
* Hello World (I am a comment)
* Spanning multiple lines.
*/
// I am also a comment

#include <iostream>

int main() {
    // Comments can go anywhere
    std::cout << "Hello, world";
    std::cout << std::endl; /* even here */
    return 0;
}
```

Preprocessor Directives

These statements are evaluated before the compilation process starts.

They begin with ‘#’

```
#include [file]

#define [key] [value]

#ifdef [key]
#ifndef [key]
#endif

#pragma [xyz]
```

```
/* Hello World */

#include <iostream>

int main() {
    std::cout << "Hello, world";
    std::cout << std::endl;
    return 0;
}
```

Before this program is compiled, everything from iostream is made available to the current file.

The things in yellow would not be available without iostream, they are *functions* from that library. Here we can also see that they are in the *standard namespace*

I/O Streams

Think of each stream as a data chute, anything you throw in will eventually get written to the corresponding output

This allows you to send multiple pieces of data at a time.

std::endl represents an end-of-line, and forces the program to flush it's buffer

Common escape characters: \n, \r, \t, \\, \”

std::cin is used to get input from users

```
/* Hello World */

#include <iostream>

int main() {
    std::cout << "Hello, world";
    std::cout << std::endl;
    return 0;
}
```

Here, “Hello, world” is being sent to the *standard output*. Anything sent before std::endl will get printed onto the same line.

Namespaces

Namespaces allow programmers to define contexts for names.

Use the *scope resolution operator* to tell the compiler which version of the cout function to use at any point.

You **can** avoid using std:: with the *using* directive. Be very careful with *using* directives, as they can cause *namespace collisions*. It is usually better to use the more specific version of *using*

```
#include <iostream>

using namespace std;

namespace mycode {
    void cout(int);
}

int main() {
    cout << "Hello, world" << endl;
    mycode::cout(5);
    return 0;
}
namespace mycode {
    void cout(int) {
        std::cout << "mycode int:" << int;
        std::cout << std::endl;
    }
}
```

main() Function

Where the program starts

Takes either 0 parameters, or 2

argc contains the number of arguments passed to the program

argv contains those arguments

Note: The first argument to argv is **always** the name of the program itself.

Return 0 == success

```
/* Hello World */

#include <iostream>

int main() {
    std::cout << "Hello, world";
    std::cout << std::endl;
    return 0;
}
int main(int argc, char* argv[]) {
    std::cout << "Hello, world";
    std::cout << std::endl;
    return 0;
}
```

Note that this code, as it is, will not compile, you can only have **one** main function!

Curly Braces

Anything between them is a unit of code, or a block

Statements are executed in the order they appear

Each block has its own scope, variables are normally discarded upon exiting a block

```
/* Hello World */

#include <iostream>

int main() {
    std::cout << “Hello, world”;
    std::cout << std::endl;
    return 0;
}
```

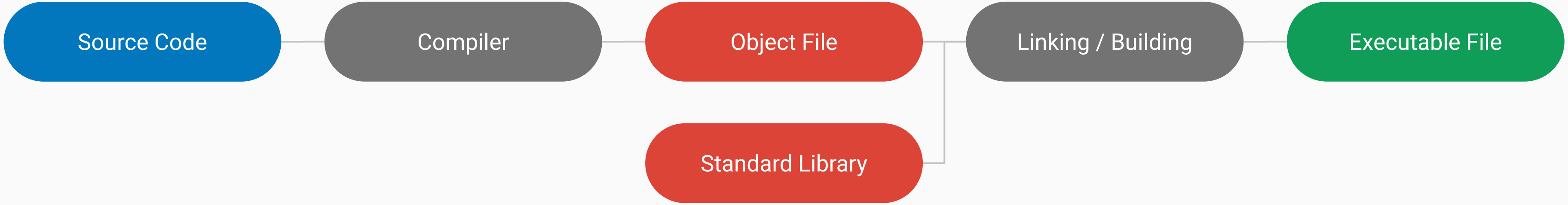
Write, Compile, and Run your own ‘Hello World’

Ask for help if you need it

Source Code -> Machine Code

Machines cannot run C++ code natively, it must be translated.

Compilers translate code in executable files:



Is the following program valid?

```
#include <iostream>
int main()   std::cout << "Hello, world!" << std::endl;
```

What about this one?

```
#include <iostream>
int main() {{{{{{
    std::cout << "Hello, world!" << std::endl;
}}}}}}
```

Write, Compile, and Run a program that writes the “Hello, World!” program to the command line.

Ask for help if you need it

Strings

String’s in use

We can use them by including the standard header <string>

In order to read input from a user we need a variable, variables always require a type in C++

std::cin reads from the standard input until it finds a non-whitespace character, then consumes characters until another whitespace is found

```
#include <iostream>
#include <string>

/* Hello, user
 * Prompts the user for their name,
 * then greet them.
 */
int main() {
    std::cout << “First name? ”;

    std::string name;
    std::cin >> name;

    std::cout << “Hello, “ << name << std::endl;
    return 0;
}
```

Using devdocs.io, modify the “Hello, user” program so that the output is as follows:

```
***************
* Hello, user *
***************
```

Hint: search for basic_string

```
#include <iostream>
#include <string>

/* Hello, user
 * Prompts the user for their name,
 * then greet them.
 */
int main() {
    // Get their name.
    std::cout << “First name? ”;
    std::string name;
    std::cin >> name;

    // Build the greeting.
    const std::string greeting = “* Hello, “ + name + “! *”;
    const std::string stars(greeting.size(), ‘*’)

    // Print it out.
    std::cout << stars << std::endl;
    std::cout << greeting << std::endl;
    std::cout << stars << std::endl;
    return 0;
}
```

Modify your program such that it prints both the first and last name of the user.

Ask for help if you need it

Variables

Variables

Variables are both declared and defines, much like functions

Undefined variables will have random values (based on whatever is in memory at that time)

```
#include <iostream>
#include <string>

int main() {
    int uninitialized_int;
    int initialized_int = 5;
    std::string text;
    const std::string more_text(8, ‘*’);
    return 0;
}
```

Types of Variables

Type	Description	Usage
int	Positive and Negative integers (4 bytes)	int i = 7;
float	Floating point numbers (4 bytes)	float f = 7.2;
double	Double precision numbers (8 bytes)	double d = 7.2;
char	A single character (1 byte)	char c = ‘a’;
bool	true or false (Non-0, 0) (1 byte)	bool b = true;
auto	The compiler will decide the type automatically (usually not recommended for declarations)	auto a = some_function()

Modifiers

C++ comes with modifiers that change the default types:

- short: applies to int; reduces the size to 2 bytes
- long: applies to int, double; increases the size to 8 bytes and 16 bytes
- long long: applies to int; might increase size
- unsigned: applies to int, char; prevents negative numbers

Internal Representation

8 bits available in a `char`:

- unsigned: all 8 bits are “counting bits”
  - range from 0 to 255.
- signed: 7 counting bits, one “sign bit” (determines negative or positive)
  - range from -128 to 127.

Variables (cont)

Variables can be converted into other types via *casting*

Explicit casting isn't always needed, variables can be *coerced* into other types.

Never implicitly cast and lose data

```
#include <iostream>

int main() {
    int initialized_int = 97;
    std::cout << (bool)initialized_int;
    std::cout << "\t";
    std::cout << (char)initialized_int;
    std::cout << std::endl;
    return 0;
}
```

Operators

Operators

Allows modification and usage of variables.

Operators can be *unary*, *binary*, or *ternary*.

```
#include <iostream>

int main() {
    int x = 5;
    std::cout << x << " + 5 = " << x + 5;
    std::cout << std::endl;
    std::cout << x / 5 + 2 * 10 - 1;
    std::cout << std::endl;
    return 0;
}
```

Operators

Operator	Description
=	Binary operator to assign the value on the right to the variable on the left
!	Unary operator to negate a variable
+, -, *, /	Binary operators for addition, subtraction, multiplication, division
%	Binary operator for the remainder of a division operation, aka *modulos*
++, --	Unary operator for increment and decrement. If these operators appear before the variable, they are called *prefix*, if they come after, they are called *postfix*.
&, |, <<, >>, ^	Binary bitwise functions: and, or, shift left, shift right, exclusive-or

All of the binary operators can also be used in their *shorthand* varieties by attaching an assignment operator to the right, ie +=, -=, %=, &=, ^=, etc

Write, Compile, and Run your own Addition Calculator

This program should prompt the user for 2 numbers, then print the answer.

Control Flow

Conditionals

Allow programs to make decisions based on some *condition*

The type of conditional shown here is called an *if-statement*

```
#include <iostream>

int main() {
    if (2 > 1) {
        std::cout << "All is good";
    } else {
        std::cout << "Something is wrong";
    }
    std::cout << std::endl;
    return 0;
}
// You can also use ternary operators
// std::cout << (2 > 1) ? true : false;
// std::cout << std::endl;
```

Conditionals cont.

If-statements may be chained together using `else if`

```
#include <iostream>

int main() {
    int a = 1; int b = 2;
    if (a > b) {
        std::cout << "a is greater";
    } else if (b > a) {
        std::cout << "b is greater";
    } else {
        std::cout << "a and b are equal";
    }
    std::cout << std::endl;
    return 0;
}
```

Conditionals cont.

If there are many possible cases, a *switch-statement* may be used

```
#include <iostream>

int main() {
    char c = 'c';
    switch (c) {
        case 'b':
        case 'a':
        case 'r':
            std::cout << "c in 'bar'";
            break;
        default:
            std::cout << "c not in 'bar'";
            break;
    }
    return 0;
}
```

Conditional Operators

<table>
<tr><th>Op</th><th>Description</th></tr>
<tr><td>&lt;, &lt;=, &gt;, &gt;=</td><td>Determines if the left-side is less than, less than or equal to, greater than, or greater than or equal to the right-hand side.</td></tr>
<tr><td>==</td><td>Determines if both sides are equal</td></tr>
<tr><td>!=</td><td>Determines if both sides are <b>not</b> equal (ie (false != true) == true)</td></tr>
<tr><td>&amp;&amp;, ||</td><td>Logical and, and logical or.</td></tr>
<tr><td>!</td><td>This operates the same as the regular ! operator, negating whatever is to its right.</td></tr>
</table>

C++ uses *short-circuit logic*, once the final result can be determined, the rest of the expression won’t be evaluated.

Modify your calculator to accept an operator

Now, your program should prompt for one number, an operator, then a second, it should then do the math, and print the answer.

Loops

Loops are used to do things repeatedly (hence *loop*).

There are a number of choices for looping:

- while
- do/while
- for
- Range-based for

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 5) {
        std::cout <<"Hello, world";
        std::cout << std::endl;
        ++i;
    }
    for (i; i>0; --i)
        std::cout << "for ";
    std::cout << std::endl;
    return 0;
}
```

REPL.it

Arrays

Arrays hold a series of values, all of the same type.

Elements are accessible via their position in the array.

Arrays have *fixed-sizes*, meaning you cannot declare them with variables

```
#include <iostream>

int main() {
    int arr[10];
    for (int i=0; i<10; ++i)
        arr[i] = i;
    for (int i=0; i<10; ++i)
        std::cout << arr[i] << ' ';
    std::cout << std::endl;
    return 0;
}

// Arrays can also be initialized as:
// int arr2[10] = {0,1,2,3,4,5,6,7,8,9};
// int arr3[10] = {0};
```

This declares an array

This initializes the values of the array

Arrays in Memory

Position	0	1	2	3	4	5	6	7	8	9	10	...
Value	0	1	2	3	4	5	6	7	8	9	10	??

Write, Compile, and Run Max Finder

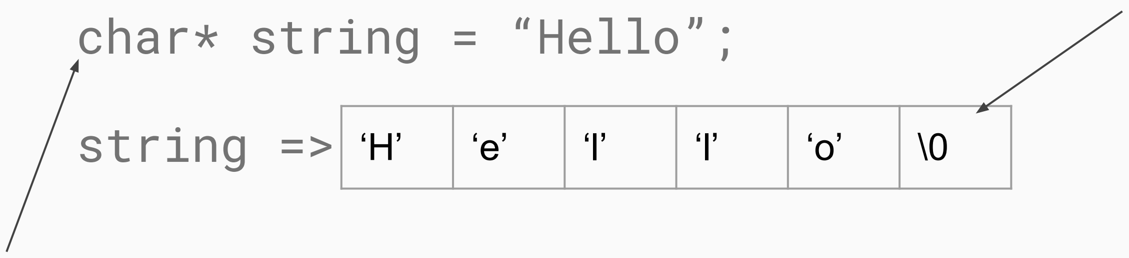
Your task is to write a program that finds the maximum value in a hard-coded array.

If you have time, try to figure out how to take input from std::cin.

C-Strings

C-Strings are simple arrays of characters

```
char* string = “Hello”;
```



`string =>`

‘H’	‘e’	‘l’	‘l’	‘o’	\0

This is the NULL character

Note the type is an array of characters, not a string

ASCII Table

Updates

- Errata
  - I had thought pragma was included in the C++11 Standard, however this is not the case. As such, we will fall back to the more portable #ifndef schema
- Homework
  - Code Signal Practice Test due at midnight. Email me a screenshot of your score
  - Github Developer Pack Application, follow the same procedure as above

Make sure you join the slack channel for the class!

Command Line Arguments

argc vs argv

argc contains the number of arguments given to the program

argv is an array of character arrays, with each array containing an argument given to the program

argv[0] is always the name of the program

```
/* Hello World */

#include <iostream>

int main(int argc, char* argv[]) {
    for (int i=0; i < argc; ++i)
        std::cout << argv[i] << “ “;
    std::cout << std::endl;
    return 0;
}
```

Exploring argc and argv

Create a program that prints out the number of arguments passed to itself.

For example:

```
$ ./count this "program counts"
$ 3
```

Modify Hello, User to use command line arguments

This program should accept a users name from the command line, not from std::cin

Abstractions

Types

You can use basic types to build more complex types of your own design.

In reality, types are not used as frequently as classes are, but they are still helpful tools to know about!

```
#include <iostream>

typedef enum {
    Monday, Tuesday, Wednesday, Thursday,
    Friday, Saturday, Sunday
} Weekday;

int main() {
    Weekday d = Monday;
    std::cout << d << std::endl;
    return 0;
}
```

Structs

Structs allow you to encapsulate one or more existing types into a new type.

This is a layer of abstraction, allowing you to group things.

```
#include <iostream>

typedef struct {
    char* first_name;
    char* last_name;
    char* email;
} Student;

int main() {
    Student s;
    s.first_name = "Tom";
    s.last_name = "Howard";
    s.email = "thoward27@uri.edu"
    std::cout << "Student:" << s.last_name;
    return 0;
}
```

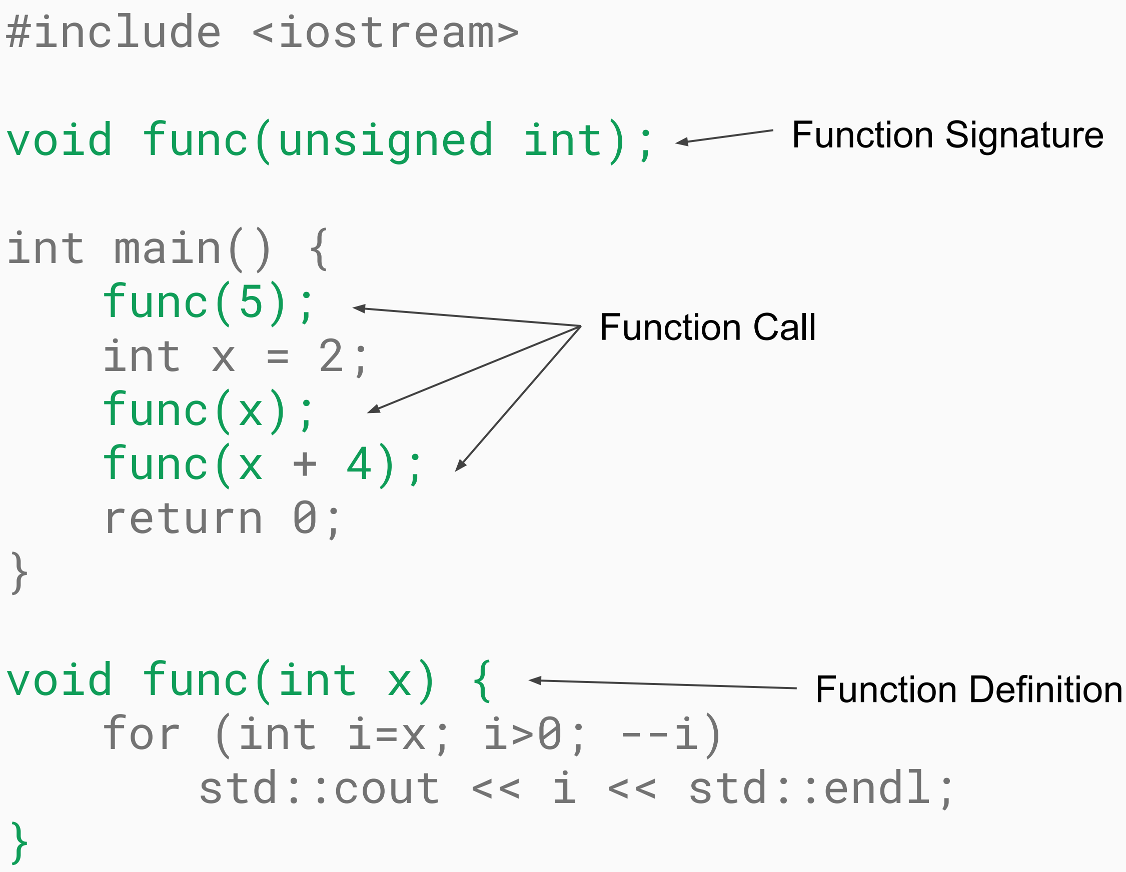
Functions

Not all code should / can go in main()!

Imagine if Microsoft Word was a single function! It would be unreadable, unmanageable, unusable!

To solve this, programmers can wrap functionality in a layer of abstraction, by making functions

Don’t forget, main is itself a function



```
#include <iostream>

void func(unsigned int);

int main() {
    func(5);
    int x = 2;
    func(x);
    func(x + 4);
    return 0;
}

void func(int x) {
    for (int i=x; i>0; --i)
        std::cout << i << std::endl;
}
```

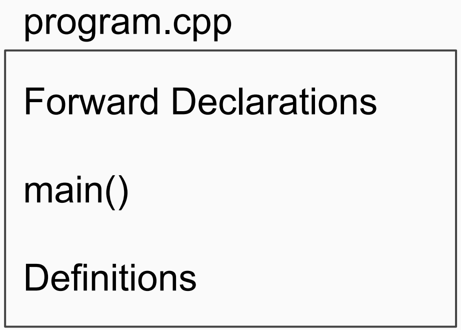
Function Signature

Function Call

Function Definition

Declarations vs Definitions

Can be done via *Forward Declaration* or via *Header Files*

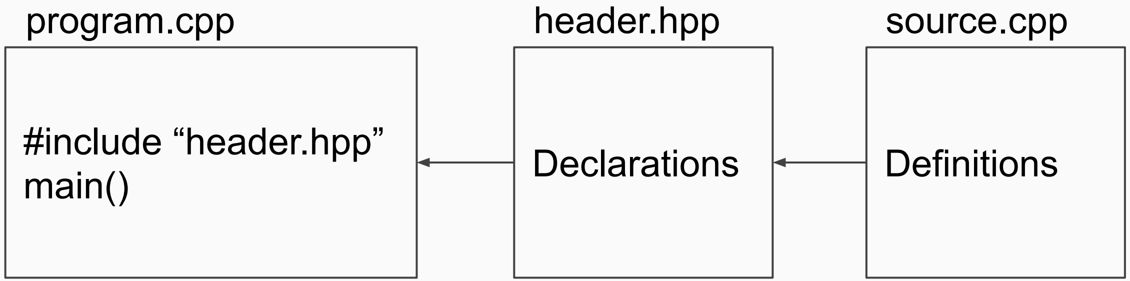


program.cpp

Forward Declarations

main()

Definitions



program.cpp

#include “header.hpp”
main()

header.hpp

Declarations

source.cpp

Definitions

Write, Compile, and Run Palindrome Tester

Your task is to write a function that can determine whether a given c-string is a palindrome or not. The function should return true or false.

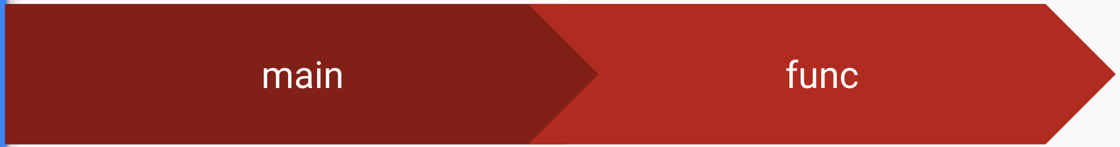
Memory Management

Stack and Heap

The stack can be visualized as a deck of cards, the top card is the current scope. As functions are called, new cards are created and placed on top with all relevant information for the function to work. As functions return, their card is *popped* off of the stack. These cards are called *stack frames*.

The heap is a place for you to throw your data. Anything placed onto the heap is persistent, and must explicitly be deleted. Data on the heap is not destroyed when functions exit!

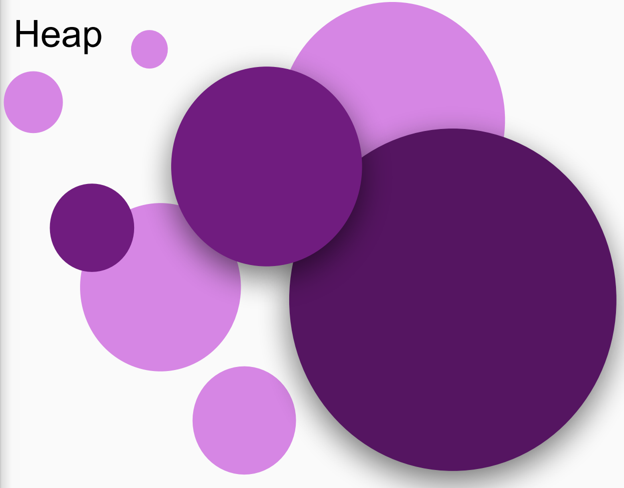
Stack

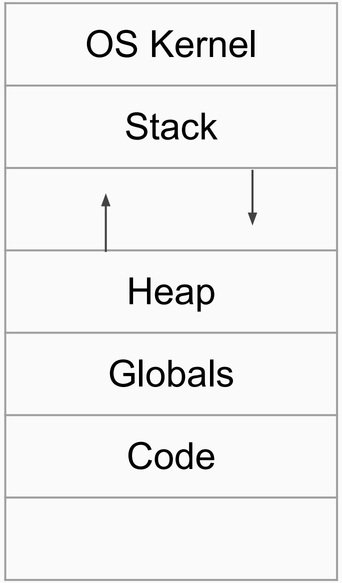


Return value: _____
x: _____

Return value: _____
x: _____
i: _____

Heap





https://goo.gl/BvXFeG

Stack vs Heap

Allocating on the stack is the default behavior, it ensures that when a function exits, all irrelevant information is freed for the OS to use

Sometimes, we do not want the memory to be freed, in that case, we must use the Heap

The heap essentially solves the problem of having a variable outlive its scope

This can be messy, but it can also be extremely powerful

Dynamically Allocated Arrays

Here `array_size` is a variable, not a constant. This is bad practice in C++

In standard C++, the compiler needs to know how much memory to allocate on the stack before runtime, but it can't!

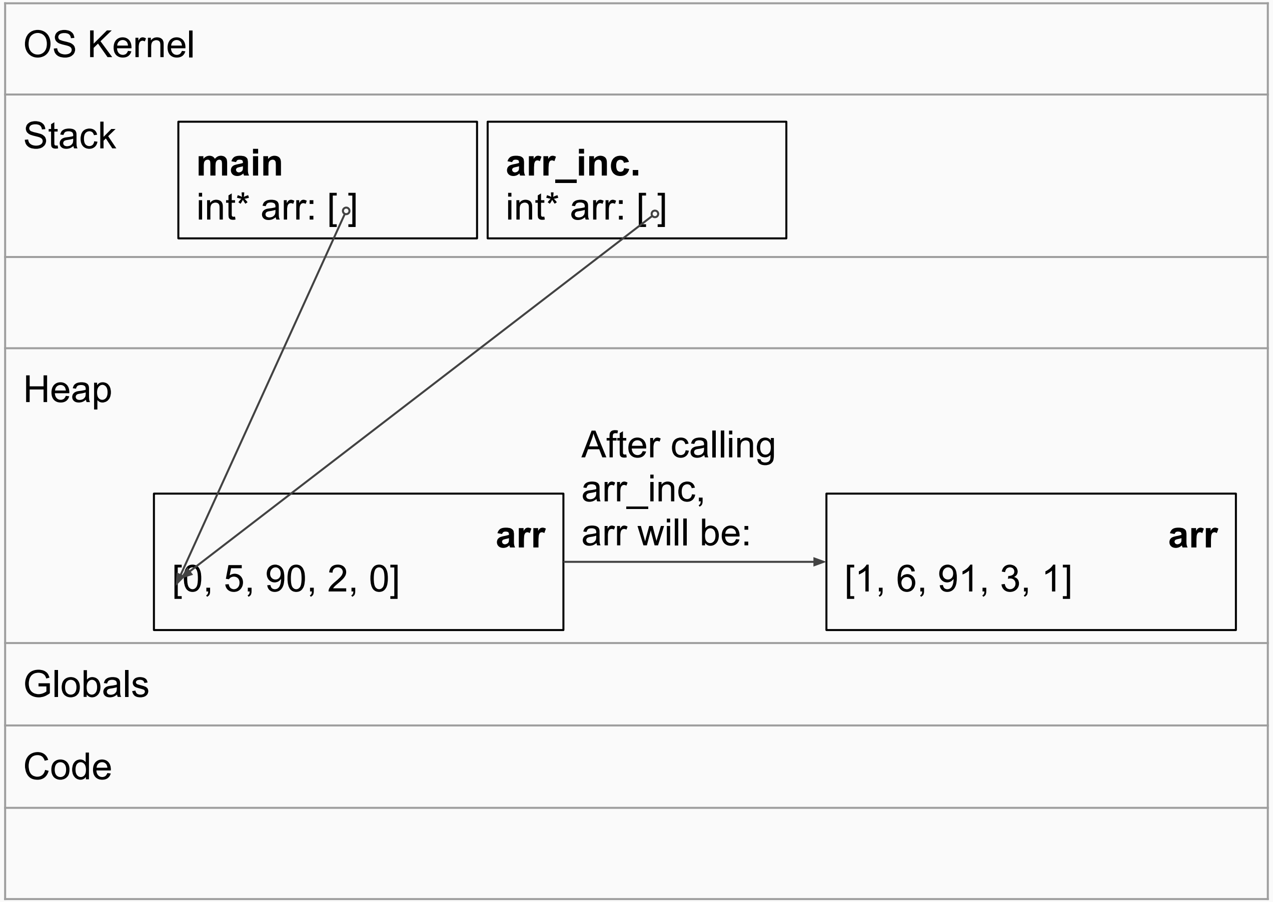
To properly allocate a variable sized array, you must first declare a *pointer*, then allocate your variable array onto the heap using *new*

```
int array_size = 8;
int array[array_size];

// ^ This is not standard!


int* variable_array;
variable_array = new int[array_size];
delete[] variable_array;

// ^ But this is!
```

Use new to allocate the array on the heap!

Use `delete` to free the memory on the heap!

Pointers



Both `main` and `arr_inc` get to work on the same array!

When control returns to the main function, the array will have been incremented

Write, Compile, and Run Array Increment

Your task is to write a program that initializes (declares and defines), an array of integers on the heap, then calls a function on that array to increment it’s values. Within `main` be sure to print the array once before, and once after the call.

Other Reasons for Pointers

Alternative to passing large arrays or other data types

Concurrency (multiple functions *can* work on the same data source)

Pointer Operators

Op	Name	Description	Example Usage
*	Dereference	Yields the value pointed to by a pointer	`int x = *p;` Assigns to x the value that p points to.
&	Address-of	Yields the memory address of a variable	`int* y = &x;` Assigns to y the memory address of the variable x.

These are **unary prefix operators**

Pointers (cont)

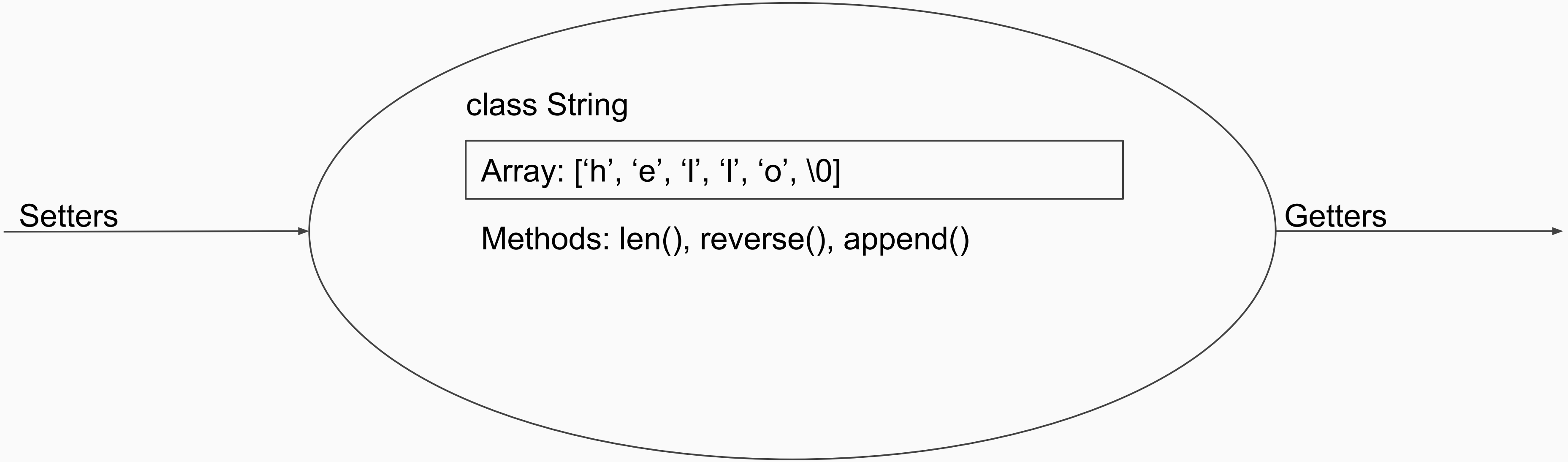
Can be used for variables on the stack, as well as for different types of variables.

```
int i = 8;
int* my_int = &i;

int* my_heap_int = new int;
*my_heap_int = 5;

int* my_heap_array = new int[5];
my_heap_array[0] = 1;

char stack_c = ‘a’;
char* my_c = &stack_c;

char* my_heap_c = new char;
*my_heap_c = ‘b’;
```

https://goo.gl/7aavK4

Classes

Classes allow data encapsulation,
the internal array is protected.



Setters

class String

Array: [‘h’, ‘e’, ‘l’, ‘l’, ‘o’, \0]

Methods: len(), reverse(), append()

Getters

String

Now we will build our own mini string class from the ground up!

We will do this mostly together, with periods of group work on specific components.

Let’s start with the class declaration

```
#ifndef __STRING__
#define __STRING__

class String {
    private:
        // Declare data members here.
    public:
        // Constructors and Destructor.
        String();
        String(const char*);
        ~String();

        // Other Functions.
        unsigned int length();
        void reverse();
        void append(const char);
};

#endif
```

Private

Only accessible from within the class itself.

Other objects cannot modify private variables.

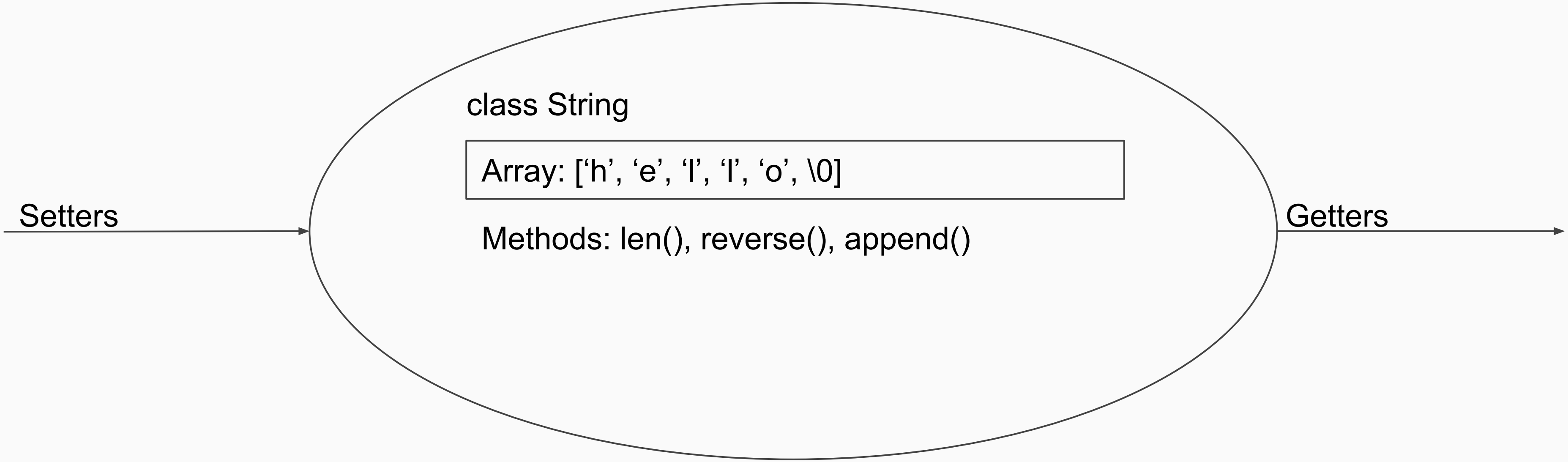
What internal data members will we need?

Should we store length?

```
class String {
    private:
        // Declare data members here.
        char * array;
        int length; // Maybe useful.
    // … Etc.
};
```

Classes

Classes allow data encapsulation,
the internal array is protected.



Setters

class String

Array: [‘h’, ‘e’, ‘l’, ‘l’, ‘o’, \0]

Methods: len(), reverse(), append()

Getters

Public

Allows interaction with the internal data

This is the API for your class

API = Application Programming Interface

```
class String {
    // … Private stuff.

    public:
        // Constructors and Destructor.
        String();
        String(const char*);
        ~String();

        // Other Functions.
        unsigned int length();
        void reverse();
        void append(const char);
};
```

Constructors & Destructors

How classes are built!

You **must** have one that takes no parameters (called default constructor)

Other constructors are optional!

Destructor must **free** all memory.

```
class String {
    // … Private stuff.

    public:
        // Constructors:
        String(); // Default
        String(const char*); // Parameterized
        ~String(); // This is the destructor

        // … other public stuff.
};
```

Write the constructors and destructor.

Ask for help if you need it

Constructors & Destructors

For the default constructor, all we need to do is initialize all of our variables to some intelligent values.

The parameterized constructor must do a bit more work.

The destructor is trivial for this class, all that needs to be done is freeing the array's memory.

```
String::String() {
    array = new char[1]; // Other values?
    array[0]  = '\0';
    length = 0;
}

String::String(const char* string) {
    unsigned int s_len = 0;
    for (int i=0; string[i]; ++i)
        ++s_len;
    array = new char[s_len + 1];
    for (int i=0; string[i]; ++i)
        array[i] = string[i];
    array[s_len] = ‘\0’;
    length = s_len;
}

String::~String() {
    delete[] array;
}
```

Implement `length()`.

Ask for help if you need it

Length

This one depends on whether or not you’ve chosen to store length as a variable or not.

What are the pros and cons to these two approaches? What would you consider when deciding between the two implementations?

```
Unsigned int String::length() {
    return length;
}

Unsigned int String::length() {
    unsigned int len = 0;
    for (int i=0; array[i]; ++i)
        ++len;
    return len;
}
```

Implement append().

Ask for help if you need it

Append

Need to remember to increase the memory!

Should we only increase by 1 every time?

Other ideas for implementation?

```
void String::append(char c) {
    char* new_array = new char[length + 1];
    for (int i=0; array[i]; ++i)
        new_array[i] = array[i];
    delete[] array;

    new_array[length++] = c;
    new_array[length] = ‘\0’
    array = new_array;
}
```

Danger! Depending on your implementation, this line may be different!

Implement `reverse()`.

Ask for help if you need it

https://repl.it/@davinbernier/MyString

Reverse

We move through the first half, swapping letters with the back half

Length is unaffected

We cannot move the NULL!

```
void String::reverse() {
    for (int i = 0; i < length / 2; i++) {
        char temp = array[i];
        array[i] = array[length - i - 1];
        array[length - i - 1] = temp;
    }
}
```

return 0;

That’s it for today.
Remember to read and study for next week!

Thanks!

Continue Learning:

Accelerated C++

SoloLearn

CodeSignal

C++ Tutorial

The C++ Programming Language

“This is a super-important quote”

- From an expert