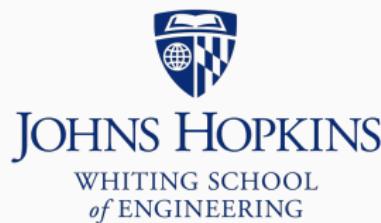


Course goals

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Course goals

Write complex, correct programs in C and C++

Leverage programming ecosystem

- Linux
- Compiler (gcc, g++)
- Debugger (gdb)
- Build system (make)

Learn basic principles of software design and engineering

Prerequisites

Experience with Java, Python or similar

C/C++ experience *not* a prerequisite

Goals

Learn C

- Language features
- Pointers & dynamic memory allocation
- “Low-level” programming

Learn C++

- How is it different from C?
- Object-oriented programming
- Generic programming

Goals

Gain proficiency in Linux & related programming tools

- Basic command-line tools
- Compilers, debuggers, profilers

Grow as a programmer & software engineer

Why C/C++?

Why C/C++?

- Ubiquitous
- Efficient
- Mature

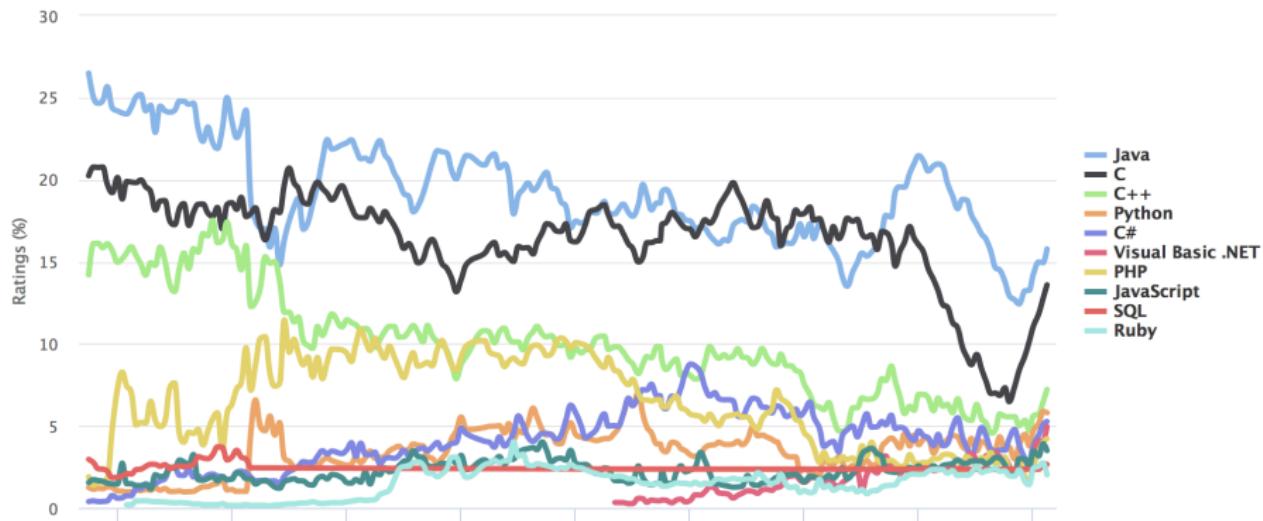
Much of the world's crucial software is in C

- Used Java? JVM is written in C++, as are many libraries
- Used Python? CPython interpreter written in C
- Used the Internet? Network stacks, routers, web servers, ...
- Like science?
 - <https://github.com/collections/software-in-science>
 - My lab members & I program in C/C++ a lot

Ubiquitous

TIOBE Programming Community Index

Source: www.tiobe.com



www.tiobe.com/tiobe-index/

Based on search engine hits for "<language> programming"

Efficient

Higher-level languages like Java & Python present a trade-off:

- High-level languages “take care of things” for you
 - Source code is more concise, abstract
 - Harder to make mistakes
- ...but also hide things from you
 - How variables are laid out in memory
 - When memory is allocated and de-allocated
 - Hardware features, especially non-portable features

Mature

Around since the 1970s (C) and 80s (C++)

Undergraduates have learned it for decades

Software jobs often require it; “we need someone who...

- ... can make something really fast if needed"
- ... knows how to program all kinds of weird hardware"
- ... knows how to interact with the operating system"
- ... can handle our large codebase, written in C"

Why C/C++?

Newer “systems languages” aim for a similar level of efficiency as C/C++

- But with simpler language
- Less “burdened” by long history & by need to stay backward-compatible

Examples

- Swift – developer.apple.com/swift
- Go – golang.org
- Rust – rust-lang.org

Why C/C++?

On efficiency, they approach C/C++

But they do not approach C/C++ in maturity/ubiquity

- Some, like Swift, are associated with (& tied to) particular companies

Why C/C++?

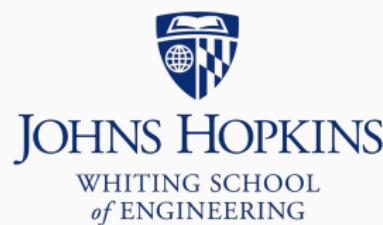
Apr 2018	Apr 2017	Change	Programming Language	Ratings	Change
1	1		Java	15.777%	+0.21%
2	2		C	13.589%	+6.62%
3	3		C++	7.218%	+2.66%
4	5	▲	Python	5.803%	+2.35%
5	4	▼	C#	5.265%	+1.69%
6	7	▲	Visual Basic .NET	4.947%	+1.70%
7	6	▼	PHP	4.218%	+0.84%
8	8		JavaScript	3.492%	+0.64%
9	-	▲	SQL	2.650%	+2.65%
10	11	▲	Ruby	2.018%	-0.29%
11	9	▼	Delphi/Object Pascal	1.961%	-0.86%
12	15	▲	R	1.806%	-0.33%
13	16	▲	Visual Basic	1.798%	-0.26%
14	13	▼	Assembly language	1.655%	-0.51%
15	12	▼	Swift	1.534%	-0.75%
16	10	▼	Perl	1.527%	-0.89%
17	17		MATLAB	1.457%	-0.59%
18	14	▼	Objective-C	1.250%	-0.91%
19	18	▼	Go	1.180%	-0.79%
20	20		PL/SQL	1.173%	-0.45%

Course conventions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Course conventions

C standard used is [C99](#)

- I will explicitly discourage use of variable length arrays

C++ standard: [C++11](#)

- I will delay introducing some C++11 features like auto and ranged-for

Editing

When I code live in the classroom, I use emacs

These slides will provide occasional emacs tips

Feel free to use your preferred editor; you might choose one with:

- Syntax highlighting (different colors denote different parts of the program)
- Integration with git
- Stronger integrations with the compiler
 - E.g. highlighting compiler warnings & errors

Code

Slides will often present source code

Where possible, code for full program will be shown:

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f", y); // print up to 2 decimal places
    return 0;
}
```

Code

When code is too long to fit, I use ... to denote elided code

```
class GradeList {  
public:  
    ...  
    void add(double grade) {  
        grades.push_back(grade);  
    }  
    ...  
private:  
    std::vector<double> grades;  
    ...  
};
```

Shell / compiling

Slides will also sometimes present shell commands:

```
$ gcc hello_world.c -std=c99 -pedantic -Wall -Wextra  
$ ./a.out  
Hello, world!
```

\$ prefixes a command you would enter at the shell prompt
(excluding the prompt \$ itself)

Output is shown just after the command that generates it

- ./a.out prints Hello, world!

Shell / compiling

Slides do not assume much Linux proficiency

- I assume you know something about *standard streams*...
 - Standard input
 - Standard output
 - Standard error
- ...and input and output redirection, e.g.:
 - `cat file.txt | ./a.out`
 - `echo "Hello world" | ./a.out`
 - `./a.out > /dev/null`

Shell / compiling

Commands and outputs are run in / transcribed from a shell session
run in a Fedora 27 Docker container

You can use the same image, but it's not necessary

- Available at: hub.docker.com/r/benlangmead/c-cpp-notes/
- Software versions:
 - gcc-7.2.1
 - g++-7.2.1
 - gdb: Fedora 8.0.1-33.fc27
 - valgrind-3.13.0
 - git v2.14.3

Shell / compiling

If you are using a different software setup than the one described on the previous slide, you may see different compiler warnings and errors than we show

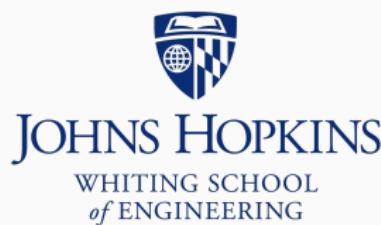
You may also experience different error messages or symptoms at runtime

Stages of compilation & Hello, world

Ben Langmead

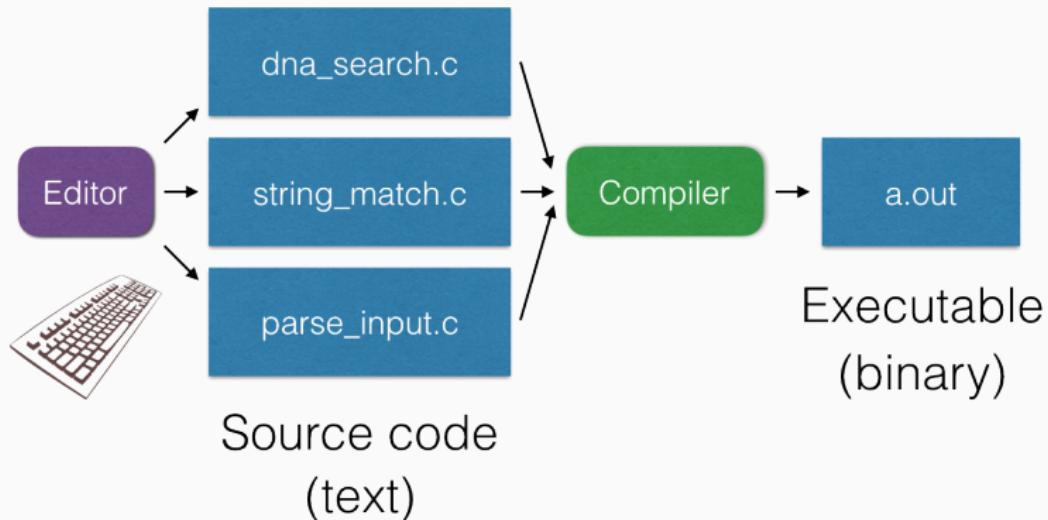
ben.langmead@gmail.com

www.langmead-lab.org

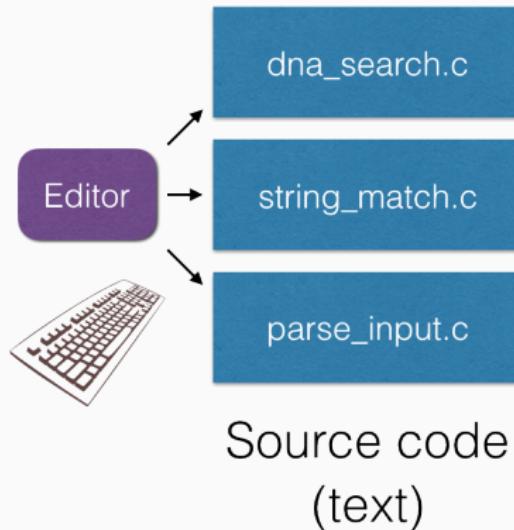


Source markdown available at github.com/BenLangmead/c-cpp-notes

Stages of compilation & Hello, world

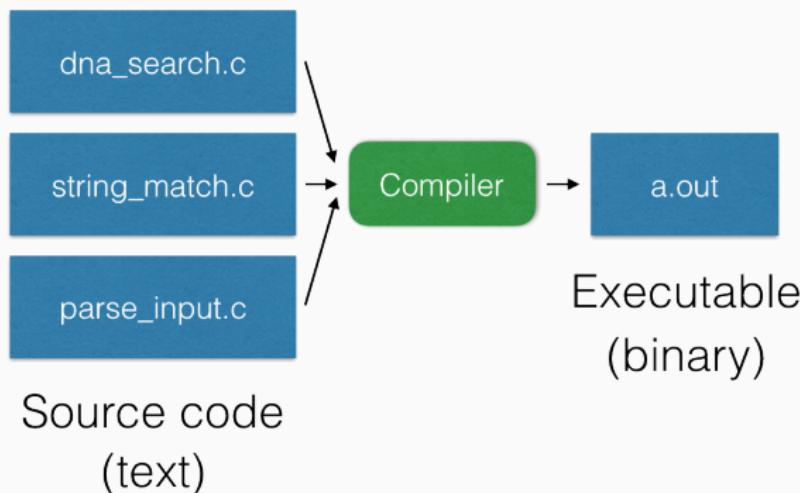


Basic C/C++ programming workflow



- emacs dna_search.c (then edit, save, exit)
- emacs string_match.c (then edit, save, exit)
- emacs parse_input.c (then edit, save, exit)

Basic C/C++ programming workflow



- `gcc dna_search.c string_match.c, parse_input.c
-std=c99 -pedantic -Wall -Wextra`

Inside the compiler

Step 1: preprocessor

- Gather relevant source code
 - Could be spread across source files that “include” & depend on each other
- Process the directives that start with #
 - We'll see `#define`, `#include`

Step 2: compiler

- Turn human-readable *source code* into *object code*
- Might yield warnings & errors if your code has mistakes

Inside the compiler

Step 3: linker

- Gather relevant *object code* into a single executable file
- Might yield warnings & errors if relevant code is missing, there's a naming conflict, etc

Inside the compiler

Between source code and machine code is a human-readable version of the machine code called “assembly language”

```
_main:                                ## @main
    .cfi_startproc
## BB#0:
    pushq   %rbp
Ltmp0:
    .cfi_def_cfa_offset 16
Ltmp1:
    .cfi_offset %rbp, -16
    movq   %rsp, %rbp
Ltmp2:
    .cfi_def_cfa_register %rbp
    subq   $16, %rsp
    leaq   L_.str(%rip), %rdi
    movl   $0, -4(%rbp)
    callq  _puts
```

Hello world

Create a directory where you will store your work for today

Go to that directory and type emacs hello_world.c

- Or use your favorite emacs alternative

Hello world

```
#include <stdio.h>

// Print "Hello world!" followed by newline and exit
int main() {
    printf("Hello world!\n");
    return 0;
}
```

After emacs hello_world.c, type above into the editor

Ctrl-x Ctrl-s to save

Ctrl-x Ctrl-c to quit

Hello world

Compile:

```
$ gcc hello_world.c -std=c99 -pedantic -Wall -Wextra  
$ ls -l a.out  
-rwxr-xr-x 1 root root 8176 May  9 11:50 a.out
```

The result is an executable program called a.out

(Later we'll see how to give it a better name)

Hello world

If a.out is in our current directory, we run it by typing ./a.out

```
$ ./a.out  
Hello world!
```

Hello world

```
#include <stdio.h>

// Print "Hello world!" followed by newline and exit
int main() {
    printf("Hello world!\n");
    return 0;
}
```

#include is a preprocessor directive

- Indicates a library to use, like import in Java or Python

main is a function, every program has exactly one main

int is its return type

() says that main takes no parameters

Hello world

```
#include <stdio.h>

// Print "Hello world!" followed by newline and exit
int main() {
    printf("Hello world!\n");
    return 0;
}
```

printf prints a string to the terminal

- \n indicates a “newline” – goes to next line

return 0 means program completed with no errors

Explanatory commenting (// Print ...) is good practice

Hello world

What if we omit #include <stdio.h>?:

```
// Print "Hello world!" followed by newline and exit
int main() {
    printf("Hello world!\n");
    return 0;
}
```

Hello world

Compiler prints a *compiler error* and does not generate a.out

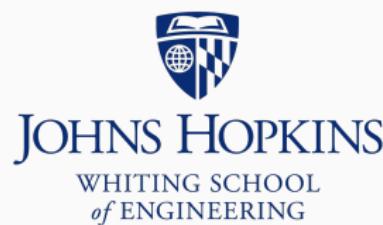
```
$ gcc hello_world_err.c -std=c99 -pedantic -Wall -Wextra
hello_world_err.c: In function 'main':
hello_world_err.c:3:5: warning: implicit declaration of function 'printf'
[-Wimplicit-function-declaration]
    printf("Hello world!\n");
    ^~~~~~
hello_world_err.c:3:5: warning: incompatible implicit declaration of built-in
function 'printf'
hello_world_err.c:3:5: note: include '<stdio.h>' or provide a declaration of
'printf'
```

Variables, types, operators

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Variables, types, operators

What does this program do?

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%.2f", y); // print up to 2 decimal places
    return 0;
}
```

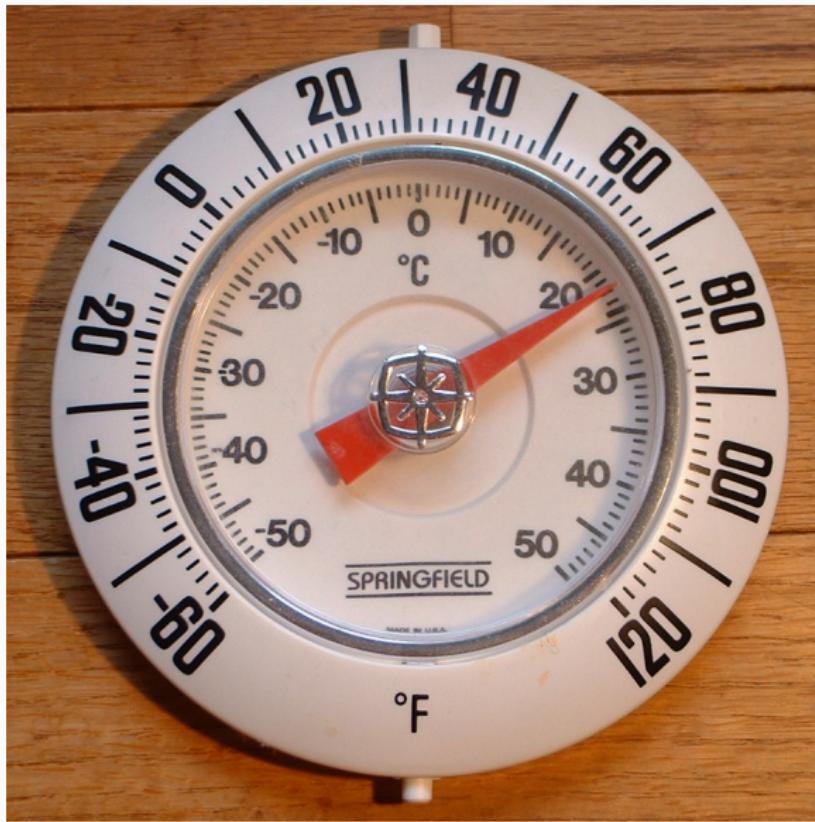
Mystery program

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%.2f", y); // print up to 2 decimal places
    return 0;
}
```

```
$ gcc mysterious.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Mysterious program



Mysterious program

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%.2f", y); // print up to 2 decimal places
    return 0;
}
```

```
$ gcc mysterious.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Mystery program

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f", y); // print up to 2 decimal places
    return 0;
}
```

- x and y are *variables*
- int and float are their *types*
- $5.0 / 9.0 * (x - 32)$ converts fahrenheit to celsius
- printf prints result as a decimal number

Variables & Assignment

```
int num_students;
```

- Variable declaration gives a *type* (`int`) and *name* (`num_students`)
 - A variable has a *value* that may change throughout the program's lifetime
- = is the *assignment operator*, which modifies a variable's value

```
num_students = 32; // assign
printf("Student added\n");
num_students = 33; // assign again
printf("2 students dropped\n");
num_students = 31; // assign again
```

Assignment

It is good practice to declare and assign *at the same time*:

```
int num_students = 32;
```

This is also called *initializing* the variable

A variable that has been declared but not yet assigned has an
“undefined” value

Mystery program

What's a good way to make this mystery program less mysterious?

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f", y); // print up to 2 decimal places
    return 0;
}
```

Less mysterious program

```
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

- Give variables meaningful names
- // explanatory comments

Types

- Integer types
 - int: signed integer
 - unsigned: unsigned integer
- Floating-point (decimal) types
 - float: single-precision floating point number
 - double: double-precision floating point number

Review

```
int num_students = 35;
```

- Variables have a name, a value and a type
- Type determines the kind of values it's allowed to have
- Good practice: give (*assign*) a variable a value at the same time as you *declare* it
- Give variables meaningful names

Types

Integer types

- int: positive or negative integer
- unsigned int: positive integer (or 0)

```
#include <stdio.h>
```

```
int main() {
    int x = -10;
    unsigned int y = 333;
    printf("%d %u\n", x, y);
    return 0;
}
```

```
$ gcc ints.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
-10 333
```

Types

Floating-point (decimal) types

- float: single-precision floating point number
- double: double-precision floating point number

```
#include <stdio.h>

int main() {
    float w = 10000.0;
    double z = 3.141592;
    printf("%f %.3f %e\n", w, z, w);
    // %e: scientific notation, %.3f: print 3 decimal places
    return 0;
}
```

```
$ gcc floats.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
10000.000000 3.142 1.000000e+04
```

Types

char: character type

- Holds a 1-byte character, 'A', 'B', '\$', '\n'...
- chars are basically integers, as we'll see

Types

Some other languages support a “Boolean” (true or false) type

In C, integers function as booleans

- 0=false, non-0=true

Types

Type	Stands for	Size*	Range
char	Character	1 byte	-128 – 127
short	Short integer	2 bytes	-32,768 – 32,767
int	Integer	4 bytes	$-2^{31} - (2^{31} - 1)$
long	Long integer	8 bytes	$-2^{63} - (2^{63} - 1)$
float	Floating point (single precision)	4 bytes	1.2e-38 – 3.4e+38
double	Floating point (double precision)	8 bytes	2.3e-308 – 1.7e+308

* Size can vary depending on platform

Types

Unsigned types can't represent negative numbers, but range to somewhat higher positive numbers:

Type	Stands for	Size*	Range
unsigned char	Unsigned character	1 byte	0 – 255
unsigned int	Unsigned integer	4 bytes	0 – 2^{32}
unsigned long	Unsigned long integer	8 bytes	0 – 2^{64}
size_t	(same)	8 bytes	0 – 2^{64}

No such thing as `unsigned float` or `unsigned double`

More on *numeric representations* and *numeric precision* later

We typically use `int`, `double` and `char`

Operators

3 + 4

- 3 and 4 are *operands*, + is *operator*
- 3 and 4 are *constants* (not variables)

num_students + 4

- num_students and 4 are operands, + is operator
- num_students is a variable

Arithmetic operators

Operator	Function	Expression	Result
+	Addition	4 + 3	7
-	Subtraction	7 - 6	1
*	Multiplication	4 * 5	20
/	Integer division	7 / 2	3
/	Floating-point division	7.0 / 2.0	3.5
%	Integer remainder	11 % 3	2

Mysterious program

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f", y); // print up to 2 decimal places
    return 0;
}
```

```
$ gcc mysterious.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Less mysterious program

```
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

- Output is correct (double-check with google)

```
$ gcc convert_fc.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Mistake?

```
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    //float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    float celsius = 5.0 / 9.0 * fahrenheit - 32;
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

Mistake?

```
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    float celsius = 5.0 / 9.0 * fahrenheit - 32 ;
    //                         was: 9.0 * (fahrenheit - 32);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}

$ gcc convert_fc_badprec.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
7.44
```

- Mistake because multiplication & division have higher *precedence* than subtraction

Operator precedence

Higher in the table means higher precedence

Operator	Function	Associativity
()	Function call / parentheses	left to right
[]	Array subscript	
. , ->	Member selection	
++, --	Postincrement/postdecrement	
++, --	Preincrement/predecrement	right to left
!	Logical negation (not)	
~	Bitwise complement (not)	
*, &	Pointer dereference, reference	
*, /, %	Multiplication, division, modulus	left to right
+, -	Addition, subtraction	left to right
<<, >>	Bitwise left- and right shift	left to right
<, <=	Relational operators	left to right
>=, >		

Operator precedence

Know where to look up the rules

- en.cppreference.com/w/c/language/operator_precedence

... and use parentheses when in doubt

Program variant

```
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    int base = 32;
    float factor = 5.0 / 9.0;
    float celsius = factor * (fahrenheit - base);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

Result is same

```
$ gcc convert_fc_var1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Using const well

```
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    const int base = 32; // can't be modified
    const float factor = 5.0 / 9.0; // can't be modified
    float celsius = factor * (fahrenheit - base);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

const

```
const int base = 32;
```

const before a type says the variable *cannot be modified later*

const

```
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    const int fahrenheit = 71;
    const float celsius = 5.0 / 9.0 * fahrenheit - 32;
    printf("%0.2f", celsius); // print up to 2 decimal places

    celsius = 5.0 / 9.0 * 07 - 32; // oops! reassigning to a const
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

const

```
$ gcc convert_fc_var3.c -std=c99 -pedantic -Wall -Wextra
convert_fc_var3.c: In function 'main':
convert_fc_var3.c:9:13: error: assignment of read-only variable 'celsius'
    celsius = 5.0 / 9.0 * 07 - 32; // oops! reassigning to a const
               ^
```

Logical & relational operators

Relational & logical operators:

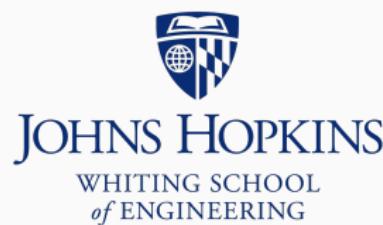
Operator	Function	Example	Result
<code>== / !=</code>	Equals / does not equal	<code>1 == 1</code>	true (non-0)
<code>< / ></code>	Less / greater	<code>5 < 7</code>	true (non-0)
<code><= / >=</code>	Less / greater or equal	<code>5 >= 7</code>	false (0)
<code>&&</code>	Both true (<i>AND</i>)	<code>1 && 0</code>	false (0)
<code> </code>	Either true (<i>OR</i>)	<code>1 0</code>	true (non-0)
<code>!</code>	Opposite (<i>NOT</i>)	<code>!(1 0)</code>	false (0)

Printing messages

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Printing messages

`printf` prints a message to screen

```
#include <stdio.h>

// Print "Hello world!" followed by newline and exit
int main() {
    printf("Hello world!\n");
    return 0;
}
```

```
$ gcc hello_world.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
Hello world!
```

`\n` at the end is a “newline” character, so subsequent output appears on the next line

Printing

printf can handle numbers and other types

```
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%.2f\n", y); // print y up to 2 decimal places
    return 0;
}
```

```
$ gcc mysterious.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Printing

`printf` is a function taking one or more *arguments*

Arguments are comma-separated & specified between parentheses

```
printf("Hello, world!\n"); // one argument
```

```
double y = 3.33;  
printf("%.2lf\n", y); // 2 arguments
```

```
printf("%d items left; price: $%.2lf\n", 10, 44.44); // 3 args
```

Printing

First argument is a *format string*

```
"%d items left; price: $%.2f"
```

Format string contain format specifiers that start with %

- Each specifies *type* of an item to print

Printing

Specifier	Type	Example output
%d	int	-43
%u	unsigned	77
%f	float	3.333333
%c	char	P
%s	string	Hello, world!

Putting l just after the % modifies the type to be longer

- %ld for long (instead of int)
- %lu for unsigned long (instead of unsigned)
- %lf for double (instead of float)

Printing

```
printf("Hello, world!\n");
```

Hello, world!

```
printf("%d\n%d\n%d\n%c\n", 3, 2, 1, '!');
```

3

2

1

!

```
printf("%f rounded is %.2f\n", 10.0/3, 10.0/3);
```

3.333333 rounded is 3.33

Printing

Questions about types?

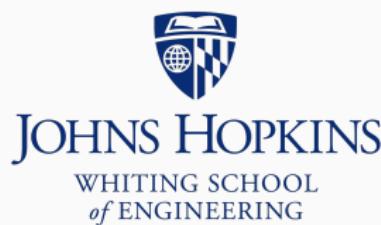
- What is the difference between int and long?
- How large are they?
- How are they represented in the computer?
- When and how can we convert between them?

We will answer these when we discuss numeric representations

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Decisions

```
#include <stdio.h>

int main() {
    int number = 12;
    if(number % 2 == 0) {
        printf("yes\n");
    }
    return 0;
}
```

Does it say “yes”?

if example

```
#include <stdio.h>

int main() {
    int number = 12;
    if(number % 2 == 0) {
        printf("yes\n");
    }
    return 0;
}
```

```
$ gcc -o if1 if1.c -std=c99 -pedantic -Wall -Wextra
$ ./if1
yes
```

if example

```
#include <stdio.h>

int main() {
    int number = 13;
    if(number % 2 == 0) {
        printf("yes\n");
    }
    else {
        printf("no\n");
    }
    return 0;
}
```

```
$ gcc -o if2 if2.c -std=c99 -pedantic -Wall -Wextra
$ ./if2
no
```

if example

```
#include <stdio.h>

int main() {
    int number = 13;
    if(number == 1) {
        printf("yes\n");
    }
    else {
        printf("no\n");
    }
    return 0;
}
```

if / else

```
$ gcc -o if3 if3.c -std=c99 -pedantic -Wall -Wextra
if3.c: In function 'main':
if3.c:5:8: warning: suggest parentheses around assignment used as truth value
[-Wparentheses]
    if(number = 1) {
        ^~~~~~
$ ./if3
yes
```

if / else

Careful: Expression `number = 1` evaluates to the assigned value, which is 1

- Makes it easier to confuse `number = 1` and `number == 1`

Fortunately, modern compilers can warn us if we mistakenly use assignment in a condition

if / else if / else

```
#include <stdio.h>
int main() {
    int x = 79;
    if(x >= 90) {
        printf("A\n");
    }
    else if(x >= 80) {
        printf("B\n");
    }
    else if(x >= 70) {
        printf("C\n");
    }
    else if(x >= 60) {
        printf("D\n");
    }
    else {
        printf("F\n");
    }
    return 0;
}
```

if / else if / else

```
$ gcc -o grading grading.c -std=c99 -pedantic -Wall -Wextra  
$ ./grading  
C
```

switch / case

```
#include <stdio.h>
int main() {
    char grade = 'C';
    int points = 0;
    switch(grade) {
        case 'A':
            points = 4;
            break;
        case 'B':
            points = 3;
            break;
        case 'C':
            points = 2;
            break;
        case 'D':
            points = 1;
            break;
        default:
            points = 0;
    }
    printf("Grade %c contributes %d GPA points\n", grade, points);
    return 0;
}
```

switch / case

```
$ gcc grading_switch.c -std=c99 -pedantic -Wall -Wextra  
$ ./a.out  
Grade C contributes 2 GPA points
```

Compound assignments

Compound assignment operators perform an operation with a variable operand and assign the result back to the variable

Example	Equivalent
<code>a += 5</code>	<code>a = a + 5</code>
<code>a -= 5</code>	<code>a = a - 5</code>
<code>a *= 5</code>	<code>a = a * 5</code>
<code>a /= 5</code>	<code>a = a / 5</code>
<code>a %= 5</code>	<code>a = a % 5</code>

Increment and decrement

Increment and decrement operators act like `+= 1` and `-= 1`

Example	Equivalent	Evaluates to
<code>a++</code>	<code>a += 1</code>	original (smaller) a
<code>++a</code>	<code>a += 1</code>	new (larger) a
<code>a--</code>	<code>a -= 1</code>	original (larger) a
<code>--a</code>	<code>a -= 1</code>	new (smaller) a

for

```
#include <stdio.h>
int main() {
    for(int i = 0; i < 10; i++) {
        printf("%d ", i);
    }
    return 0;
}
```

```
$ gcc -o for_example for_example.c -std=c99 -pedantic -Wall -Wextra
$ ./for_example
0 1 2 3 4 5 6 7 8 9
```

while

```
#include <stdio.h>
int main() {
    int i = 1;
    while((i % 7) != 0) {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

```
$ gcc while_example.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
1 2 3 4 5 6
```

do / while

```
#include <stdio.h>
int main() {
    int i = 0;
    do {
        printf("%d ", i);
        i++;
    } while((i % 7) != 0);
    return 0;
}
```

```
$ gcc do_while_example.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
0 1 2 3 4 5 6
```

scanf loop

```
#include <stdio.h>
int main() {
    int sum = 0;
    while(1) {
        int addend = 0;
        if (scanf("%d", &addend) != 1) {
            break; // immediately exit loop
        }
        sum += addend;
    }
    printf("%d\n", sum);
    return 0;
}
```

This continues to scan even when you press enter. To signal end-of-input, press Ctrl-D (possibly twice).

Nesting

Loops can be nested inside other loops

Loops & decision statements can be arbitrarily nested

- We use terms *outer* & *inner* to distinguish levels of nesting

```
while(...) {           // OUTER LOOP
    if(...) {
        do {           // INNER LOOP
            if(...) {
                ...
            } else {
                ...
            }
        } while(...);
    }
}
```

break & continue

break exits a loop, continue advances to the next iteration

They affect the *most immediate loop* containing the break/continue.

```
while(...) {           // OUTER LOOP
    if(...) {
        break;         // exit outer loop
    }
    do {             // INNER LOOP
        if(...) {
            continue; // advance inner loop
        }
    } while(...);
}
```

break/continue work *only* with loops, not with if/else, functions, etc

Code blocks

In examples so far, we used `{...}` to delimit a block of code:

```
while(...) {  
    if(...) {  
        break;  
    }  
    do {  
        if(...) {  
            continue;  
        }  
    } while(...);  
}
```

Code blocks

If block consists of single statement, can omit { and }

```
while(...) {  
    if(...) break;          // { } omitted  
    do {  
        if(...) continue; // { } omitted  
    } while(...);  
}
```

Loop summary

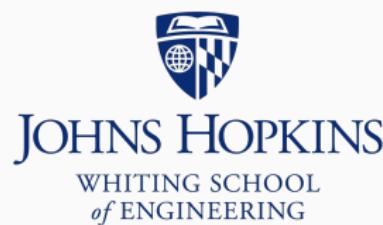
- `while(boolean expression) { statements }`
 - Iterates ≥ 0 times, as long as expression is true
- `do { statements } while(boolean expression)`
 - Iterates ≥ 1 times; always once, then more times as long as expression is true
- `for(initialize; boolean exp; update) { stmts }`
 - initialize happens first; usually declares & assigns “index variable”
 - Iterates ≥ 0 times, as long as boolean expression is true
 - Right after `stmts`, update is run; often it increments the index variable (`i++`)
- `break` immediately exits loop
- `continue` immediately proceeds to next iteration of loop

Arrays

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Arrays

```
int c[12];
```

An *array* variable consists of several elements laid out consecutively in memory

All elements have same type; int in this example

Individual elements accessed with [] notation

[0] refers to first element

Arrays

```
#include <stdio.h>
int main() {
    int c[12];
    c[0] = 7; // first element
    c[11] = 1; // last element
    printf("first c=%d, last c=%d\n", c[0], c[11]);
    return 0;
}
```

```
$ gcc array_1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
first c=7, last c=1
```

Arrays

Square brackets go after the variable name, not after the type

- Unlike Java!

```
int main() {  
    int[12] c; // oops  
    return 0;  
}
```

```
$ gcc array_2.c -std=c99 -pedantic -Wall -Wextra  
array_2.c: In function 'main':  
array_2.c:2:8: error: expected identifier or '(' before '[' token  
    int[12] c; // oops  
           ^
```

Arrays

Danger: Elements are undefined until explicitly initialized

One way to initialize is with a loop:

```
#include <stdio.h>
int main() {
    int c[12];
    for(int i = 0; i < 12; i++) {
        c[i] = i;
    }
    printf("c[4]=%d, c[9]=%d\n", c[4], c[9]);
    return 0;
}
```

```
$ gcc array_3.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
c[4]=4, c[9]=9
```

Arrays

Can initialize to a specified sequence of values

Comma separated within { ... }:

```
#include <stdio.h>
int main() {
    int c[5] = {2, 4, 6, 8, 10};
    printf("c[1]=%d, c[3]=%d\n", c[1], c[3]);
    return 0;
}
```

```
$ gcc array_4.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
c[1]=4, c[3]=8
```

Arrays

When initializing with { ... }, array size can be omitted

Compiler figures it out

```
#include <stdio.h>
int main() {
    int c[ ] = {2, 4, 6, 8, 10};
    //      ^ no size
    printf("c[1]=%d, c[3]=%d\n", c[1], c[3]);
    return 0;
}
```

```
$ gcc array_5.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
c[1]=4, c[3]=8
```

Arrays

```
#include <stdio.h>
int main() {
    int data[10] = {2, 1, 1, 1, 2, 0, 1, 2, 1, 0};
    int freq[3] = {0, 0, 0};
    for(int i = 0; i < 10; i++) {
        freq[data[i]]++;
    }
    printf("%d, %d, %d\n", freq[0], freq[1], freq[2]);
    return 0;
}
```

```
$ gcc array_6.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
2, 5, 3
```

What would happen if some elements of data were 3?

Arrays

Typical array-related mistakes:

- Failing to initialize
 - Element values are unpredictable (*undefined*) until initialized
- Trying to access an element *out of bounds*
 - Later we will diagnose using valgrind

Arrays

```
#include <stdio.h>
int main() {
    int data[10] = {2, 3, 3, 1, 2, 0, 1, 2, 1, 0};
    int freq[3] = {0, 0, 0};
    for(int i = 0; i < 10; i++) {
        freq[data[i]]++; // oops! we go out of bounds when data[i] is 3
    }
    printf("%d, %d, %d\n", freq[0], freq[1], freq[2]);
    return 0;
}
```

```
$ gcc array_7.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
2, 3, 3
```

Going out of bounds might not yield an explicit error (unlike Java & Python) and might not crash

Arrays

```
#include <stdio.h>
int main() {
    int data[10] = {2, 3, 3, 1, 2, 1000000, 1, 2, 1, 0};
    //                                     !!! ^^^^^^^^ !!!
    int freq[3] = {0, 0, 0};
    for(int i = 0; i < 10; i++) {
        freq[data[i]]++;
        // oops! we can go *way* out of bounds
    }
    printf("%d, %d, %d\n", freq[0], freq[1], freq[2]);
    return 0;
}
```

```
$ gcc array_8.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
```

Segmentation fault: 11

Going farther out of bounds increases chance that program will crash 11

Arrays

`sizeof(x)` returns the number of bytes used to store `x` as an `unsigned long`

```
#include <stdio.h>
int main() {
    double d_single;
    double d_array10[10];
    printf("sizeof(d_single) = %lu\n", sizeof(d_single));
    printf("sizeof(d_array10) = %lu\n", sizeof(d_array10));
    return 0;
}
```

```
$ gcc sizeof.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
sizeof(d_single) = 8
sizeof(d_array10) = 80
```

Arrays

Unlike Python, no “slicing” in C

- E.g. can't access several elements at once using `c[1:4]`

More discussion of arrays later:

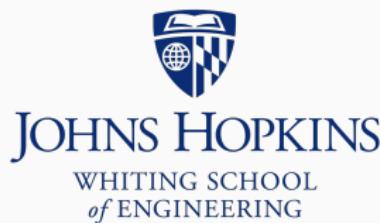
- How to pass them to and from functions
- Their relationship to pointers

Characters & strings

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Characters & strings

char variable holds a single character

- `char digit = '4';`
- `char bang = '!';`
- These *must* be single quotes; double quotes are for strings only

Behind the scenes, char is much like int

- This is valid: `char digit = '4' - 1;`
- `digit` now contains the character '`3`'

`printf` format string for char is `%c`

ASCII governs the mapping between typical characters and integers

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	'
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	-	127	7F	[DEL]

commons.wikimedia.org/wiki/File:ASCII-Table-wide.svg

Character example

```
#include <stdio.h>

// Convert decimal character into corresponding int
int main() {
    char char_0 = '0';
    int int_0 = char_0 - '0';
    printf("Character printed as character: %c\n", char_0);
    printf("Character printed as integer: %d\n", char_0);
    printf("Integer printed as integer: %d\n", int_0);
}

$ gcc convert_digit_0.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
Character printed as character: 0
Character printed as integer: 48
Integer printed as integer: 0
```

Character example

```
#include <stdio.h>

// Convert decimal character into corresponding int
int main() {
    char char_7 = '7';
    int int_7 = char_7 - '0';
    printf("Character printed as character: %c\n", char_7);
    printf("Character printed as integer: %d\n", char_7);
    printf("Integer printed as integer: %d\n", int_7);
}

$ gcc convert_digit_7.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
Character printed as character: 7
Character printed as integer: 55
Integer printed as integer: 7
```

Strings

Variety of real-word data is represented as strings:

- Natural language: "Hello world"
- DNA: "GATCACAGGTCTATCACCCCTATTAAACCACTCACGGGAGC"
- Stock movements: "BAC -0.3 GOOGL +0.8 CIT -1.2"
- Chess moves: "1.e4 e5 2.Nf3 f6 3.Nxe5 fxe5"

Strings

In C, a string is an array of characters with final character equal to the “null character” '\0'

To declare string:

```
char day[] = "monday";
```

```
char *day_ptr = "monday";
```

```
const char cday[] = "monday";
```

```
const char *cday_ptr = "monday";
```

```
// can't modify characters in cday or cday_ptr
```

A string in double quotes (e.g. "monday") is a *string literal*

Strings

These declarations show that strings have a “pointer nature” and an “array nature”:

```
char day[] = "monday";
```

```
char *day_ptr = "monday";
```

These are largely interchangeable; some differences covered later

Strings

A C string is an array of characters with the final character equal to the “null character” '\0'

Also called the the *null terminator*

```
// this definition:  
char day1[] = "monday";
```

```
// is the same as this:  
char day2[] = {'m', 'o', 'n', 'd', 'a', 'y', '\0'};  
//                                         terminator -> ^^^^
```

A character in single quotes (e.g. 'm') is a *character literal*

Strings

Access the characters in a string using square brackets, same as arrays:

```
#include <stdio.h>

// Convert decimal character into corresponding int
int main() {
    char str[] = "hello";
    str[0] = 'H'; // modify first character
    // print whole string, then 3rd char, then 5th
    printf("%s %c %c\n", str, str[2], str[4]);
    return 0;
}

$ gcc string_indexing_1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
Hello l o
```

Strings

To print string with printf use %s format specifier

```
#include <stdio.h>

// Convert decimal character into corresponding int
int main() {
    const char str[] = "World";
    printf("Hello, %s!\n", str);
    return 0;
}
```

```
$ gcc string_indexing_1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
Hello, World!
```

Strings

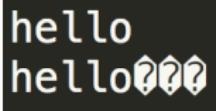
What's the mistake?

```
#include <stdio.h>

int main() {
    const char str[] = "hello";
    char str_copy[5];
    for(int i = 0; i < 5; i++) {
        str_copy[i] = str[i];
    }
    printf("%s\n", str);
    printf("%s\n", str_copy);
    return 0;
}
```

Strings

```
gcc %PREV% -std=c99 -pedantic -Wall -Wextra  
./a.out
```



```
hello  
hello???
```

Failed to null-terminate the copy, so printf prints beyond the end

- That's memory we don't own and didn't initialize
- It's “undefined”; could be anything!
- In this case, it's “garbage” that prints out as question marks

Strings

```
#include <stdio.h>

int main() {
    const char str[] = "hello";
    char str_copy[6]; // was [5]
    for(int i = 0; i < 6; i++) { // was i < 5
        str_copy[i] = str[i];
    }
    printf("%s\n", str);
    printf("%s\n", str_copy);
    return 0;
}

$ gcc string_copy_2.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
hello
hello
```

Strings

Long string can be declared across multiple lines:

```
// beginning of human mitochondrial sequence
const char *dna = "GATCACAGGTCTATCACCTATTAA"
                  "CCACTCACGGGAGCTCTCCATGCAT"
                  "TTGGTATTTCGTCTGGGGGTGTG"
                  "CACGCGATAGCATTGCGAGACGCTG"
                  "GAGCCGGAGCACCTATGTCGCAGT"
                  "ATCTGTCTTGATTCCCTGCCTCATT"
                  "CTATTATTTATCGCACCTACGTTCA"
                  "ATATTACAGGCGAACATACTACTA"
                  "AAGTGTGTTAATTAATTAATGCTTG"
                  "TAGGACATAATAACAATTGAAT";
```

Strings

```
#include <string.h> for helpful string functions
```

```
size_t strlen(const char *s);
```

Returns length of string s *not including* terminator

```
#include <stdio.h> // for printf
#include <string.h> // for strlen
```

```
int main() {
    const char str[] = "hello";
    printf("length(%s) = %lu\n", str, strlen(str));
    return 0;
}
```

```
$ gcc strlen_1.c -std=c99 -pedantic -Wall -Wextra
```

```
$ ./a.out
```

```
length(hello) = 5
```

Strings

More <string.h> functions

- `strlen(s)` returns length of string `s`
- `strcmp(s1, s2)` compares two strings alphabetically
 - negative: `s1` alphabetically before `s2`
 - zero: `s1` and `s2` equal
 - positive: `s2` alphabetically before `s1`
- `strncpy(dst, src)` copies characters from `src` into `dst`
- `strncat(dst, src)` copies `src` onto the end of `dst`
- `strstr(lng, shrt)` search for occurrence of `shrt` within `lng`

<http://www.cplusplus.com/reference/cstring/>

Strings

Some C string functions are considered unsafe and *should not* be used:

- `strcat` (instead we use `strncat`)
- `strcpy` (instead we use `strncpy`)

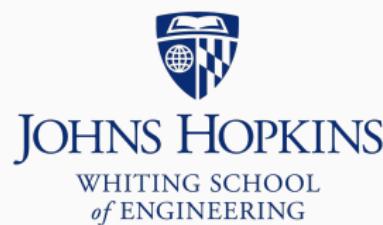
[Common vulnerabilities guide for C programmers](#)

Reading input

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Reading input

`printf` writes formatted output strings

`scanf` reads formatted *input* strings

Both use “format tags” we’ve already seen

- `%d`: int
- `%u`: unsigned int
- `%f`: float or double
- `%.3f`: float / double, up to 3 decimal places
- `%e`: float / double, scientific notation

scanf

```
#include <stdio.h>
int main() {
    int a = 0;
    float b = 0;
    // read an integer, and a floating-point
    // number from stdin, separated by space
    scanf("%d%f", &a, &b);
    printf("Read in a=%d, b=%.1f\n", a, b);
    return 0;
}
```

```
$ gcc formatted_io_eg1.c -std=c99 -pedantic -Wall -Wextra
$ echo "7 99.9" | ./a.out
Read in a=7, b=99.9
```

scanf

`scanf("%d%f", &a, &b)`: read an integer and a floating-point number separated by whitespace; store integer in `a` and floating-point in `b`

“Whitespace” means spaces, tabs & newlines. `scanf` allows any whitespace characters to separate items.

We write `&a`, `&b` instead of `a`, `b` because `scanf` *modifies* them

- `&` means “address-of”; more on this when we discuss pointers

scanf

What does echo "7 99.9" | ./a.out do?

```
$ gcc formatted_io_eg1.c -std=c99 -pedantic -Wall -Wextra  
$ echo "7 99.9" | ./a.out  
Read in a=7, b=99.9
```

Prints “7 99.9” and runs ./a.out, sending “7 99.9” to its “standard in” file

scanf

scanf returns number of items successfully parsed

It's good practice to *check this*; if it's not what you expect, something's wrong:

```
// if result is not 2, something unexpected happened
if (scanf("%d%f", &a, &b) != 2) {
    puts("Parsing error");
    return 1; // non-zero return value
}
```

scanf

Reading a string with `scanf("%s", array)` is *unsafe* because the string may be longer than the array

```
#include <stdio.h>

int main() {
    // I *think* the string will be no longer than
    // 10 chars (incl. terminator)
    char str[10];
    int nread = scanf("%s", str);
    if(nread != 1) {
        puts("Parsing error");
        return 1;
    }
    return 0;
}
```

scanf

This is fine:

```
$ gcc scanf_str.c -std=c99 -pedantic -Wall -Wextra  
$ echo "hello" | ./a.out
```

This overflows the buffer:

```
$ echo "hellohellohello" | ./a.out
```

Unpredictable things can happen when you overflow a buffer

- On my computer, it crashes
- Can also result in security vulnerabilities

scanf

Instead, include a number between % and s to put a limit on how many characters to read

```
#include <stdio.h>

int main() {
    char str[10];
    int nread = scanf("%9s", str);
    //           ^^^
    if(nread != 1) {
        puts("Parsing error");
        return 1;
    }
    printf("I read: %s\n", str);
    return 0;
}
```

scanf

```
$ gcc scanf_str2.c -std=c99 -pedantic -Wall -Wextra  
$ echo "hellohellohello" | ./a.out  
I read: hellohell
```

"%9s" means "read the string up to the ninth character"

We use "%9s" instead of "%10s" because the number doesn't count the null terminator, which scanf also writes

scanf

More scanf details:

en.wikipedia.org/wiki/Scanf_format_string

Character-by-character

`putchar` writes a single character to standard output

`getchar` reads a single character from standard input

Character-by-character

```
#include <stdio.h>
#include <ctype.h>

int main() {
    // read character-by-character until we get newline
    int c = 0;
    while((c = getchar()) != '\n') {
        // write uppercase version of character
        putchar(toupper(c));
    }
    return 0;
}

$ gcc cap_chars.c -std=c99 -pedantic -Wall -Wextra
$ echo "hello" | ./a.out
HELLO
```

Line-by-line

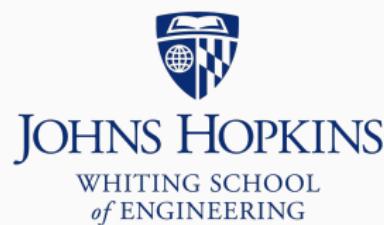
We'll defer discussion of line-by-line input until we discuss reading and writing from files

Command line arguments

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Command line arguments

We talked about how to read input from standard in

- `scanf` & `getchar`

Another way to get input is with *command-line arguments*

Command-line arguments

You are used to running programs with command-line arguments:

- `mkdir cs220` – makes a directory called `cs220`
- `ls *.txt` – lists all the files ending in `.txt`
- `cd $HOME/programming` – changes to the `$HOME/programming` directory

C programs can take command-line arguments, but we must tell the main function how to receive them

Command-line arguments

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("argc = %d\n", argc);
    for(int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

```
$ gcc args_eg_1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out rosebud
argc = 2
argv[0] = ./a.out
argv[1] = rosebud
```

Command-line arguments

```
main(int argc, char* argv[])
```

- int argc equals the number of command-line arguments.
Program name (e.g. ./a.out) is always included first.
- char* argv[] is an array of strings, one per command-line argument.
 - Sometimes written as char **argv; means the same thing

Command-line arguments

```
#include <stdio.h>
#include <stdlib.h> // for atof

int main(int argc, char* argv[]) {
    double total = 0.0;
    // i = 1 skips first argument, which is ./a.out
    for(int i = 1; i < argc; i++) {
        // atof converts a string to a double
        total += atof(argv[i]);
    }
    printf("Total: %.1f\n", total);
    return 0;
}
```

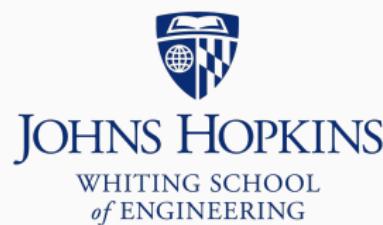
```
$ gcc args_eg_2.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out 1.1 2.2 3.3
Total: 6.6
```

Assertions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Assertions

Help to:

- Catch bugs as close to the source as possible
- Document and enforce *assumptions* and *invariants*

Assertions

```
#include <assert.h>

assert(true/false expression);
```

If boolean expression is false:

- Program exits immediately
- Exit level is non-zero, like when main returns non-zero
- Message is printed like:

```
Assertion failed: (denom != 0), function main, file foo.c, line 95.
```

Assertions

Document your assumptions:

```
int sum = a*a + b*b;  
assert(sum >= 0);
```

```
if(isalpha(c)) {  
    assert(c >= 'A');  
    printf("%d\n", c - 'A');  
}
```

Assertions

assert is not for typical error checking

```
FILE* input = fopen("numbers.txt", "r");
if(input == NULL) {
    printf("Error: could not open input file\n");
    return 1; // indicate error
}
```

Assertions

If checking for bad user input, or something else that is *strange but not impossible*:

- Use if,
- print a meaningful message, and
- if you must exit, return non-zero from main to indicate failure

For conditions that imply *your program is incorrect*, use assert

Assertions

```
#include <stdio.h>
#include <assert.h>

int main() {
    int n = 0;
    scanf("%d", &n);
    if(n == 0) {
        printf("n must not be 0\n");
        return 1;
    }
    int n_sq = n * n;
    assert(n_sq >= n); // if false, something's wrong
    float n_inv = 1.0 / n;
    printf("squared=%d, inverse=%0.2f\n", n_sq, n_inv);
    return 0;
}
```

Assertions

```
$ gcc assert_eg.c -std=c99 -pedantic -Wall -Wextra
$ echo 4 | ./a.out
squared=16, inverse=0.25

$ echo -2 | ./a.out
squared=4, inverse=-0.50

$ echo 0 | ./a.out
n must not be 0

$ echo 200000000 | ./a.out
Assertion failed: (n_sq >= n), function main,
file assert_eg.c, line 12.
```

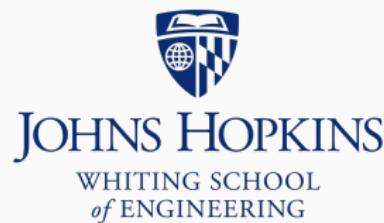
Due to overflow of int!

- We'll talk more about overflow later

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Math functions

```
#include math.h and compile with -lm option to gain access to  
these basic mathematical functions:
```

- `sqrt(x)`: square root
- `pow(x, y)`: x^y
- `exp(x)`: e^x
- `log(x)`: natural log
- `log10(x)`: log base 10
- `ceil(x) / floor(x)`: round up / down to nearest integer
- `sin(x)`: sine (other trigonometric functions available)

Math functions

x and y arguments have type double

It's also OK to pass another numeric type, like int

- Argument type promotion: int -> float -> double
- lm includes the math library when *linking*

Math functions

```
#include <stdio.h>
#include <math.h>

int main() {
    // a and b are the short side lengths for right triangle
    // Pythagorean theorem: a*a + b*b = c*c
    float a, b;
    scanf("%f%f", &a, &b);
    float c = sqrt(a*a + b*b);
    printf("Third side length = %.3f\n", c);
    return 0;
}

$ gcc assert_eg.c -lm -std=c99 -pedantic -Wall -Wextra
$ echo 3.0 4.0 | ./a.out
Third side length = 5.000

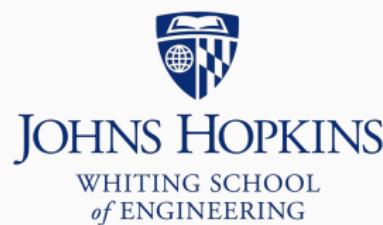
$ echo 3.0 3.0 | ./a.out
Third side length = 4.243
```

Compiling and linking

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Compiling and linking

We've seen:

- How to compile with gcc, creating an executable a.out
- How to use libraries by adding #include to the source
- How to use libraries by adding -l to the gcc command
 - E.g. -lm for math library
- The difference between *declaring* and *defining* a variable

These ideas come into sharper focus when we think about all that happens when we run gcc

Compiling and linking

What gcc does can be divided into three phases:

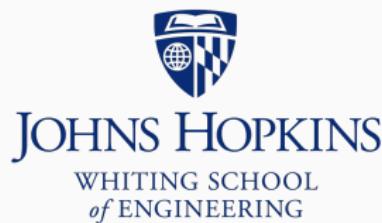
1. *Preprocessing*: bring all the relevant code together
2. *Compiling*: turn the human-readable source code into machine-readable object files
3. *Linking*: bring all relevant object files together into a binary executable

Functions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Functions

So far we have written our programs so that all the functionality is in the `main` function

Real-world programs are spread over many functions; this:

- Helps the programmer focus on smaller problems, one at a time
- Improves readability
- Helps with testing
 - Can test *functions* one by one, as we'll see
- Easier to collaborate
 - "Alice will write function X, Bob will write function Y, Carol will write Z assuming X and Y exist."

Functions

Putting a chunk of code in a new function is appropriate when:

- The code has a clear, distinct goal
- The number of variables used is not large
- Has one result (or a few)

Function examples

```
int add(int lhs, int rhs) {  
    int sum = lhs + rhs;  
    return sum; // 2. returns from add  
}  
  
int main() {  
    printf("%d\n", add(4, 5)); // 1. calls add  
    return 0; // 3. returns from main, exiting program  
}
```

Here: `main` is the *caller*, `add` is the *callee*

`return` exits `add`, sending the `return` value to *callee*

`return` from `main` is special: exits the program

Functions

```
float fahrenheit_to_celcius(float fahrenheit) {  
    float scale = 5.0 / 9.0;  
    return scale * (fahrenheit - 32);  
}
```

Function can have its own *local* variables (scale)

fahrenheit is a *parameter*, and also a local variable

When function returns, local variables “disappear”

Functions

```
#include <stdio.h>

int sum(int a, int b) {
    int c = a + b;
    // c "disappears" when function returns
    return c;
}

int main() {
    sum(7, 5);
    // error: there's nothing called "c"
    printf("%d\n", c);
}
```

Functions

```
$ gcc function_eg2.c -std=c99 -pedantic -Wall -Wextra
function_eg2.c: In function 'main':
function_eg2.c:12:20: error: 'c' undeclared (first use in this function)
    printf("%d\n", c);
                           ^
function_eg2.c:12:20: note: each undeclared identifier is reported only once for
each function it appears in
```

Functions

There are two variables called c here; changing one doesn't affect other

```
#include <stdio.h>

int sum(int a, int b) {
    int c = a + b; // doesn't affect 'c' in main
    return c;
}

int main() {
    int c = 0;
    sum(7, 5);
    printf("%d\n", c);
    return 0;
}

$ gcc function_eg3.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
0
```

Functions

```
int sum(int a, int b) {  
    int c = a + b;  
    return c;  
}
```

In C, parameters are passed *by value*

Callee's parameter receives a *copy* of the value

In general, changes to local variables (including parameters) in the callee are *not* reflected in the caller

Functions

```
#include <stdio.h>

void assign_7(int a) { // different `a` from the one in main
    a = 7;
}

int main() {
    int a = 0;
    assign_7(a);
    printf("%d\n", a); // main's `a` is unchanged
    return 0;
}
```

Functions

Compiler warns us in this case:

```
$ gcc function_eg4.c -std=c99 -pedantic -Wall -Wextra
function_eg4.c: In function 'assign_7':
function_eg4.c:3:19: warning: parameter 'a' set but not used [-Wunused-but-set-
parameter]
void assign_7(int a) { // different `a` from the one in main
    ^
$ ./a.out
0
```

Functions

An alternative to passing by value is to pass *by pointer*, which allows callee to modify a variable local to the caller

We've done this: `scanf("%f%f", &a, &b)`

We'll return to this when we talk about pointers

Passing arrays & strings is a special topic that we cover a little later

Functions

Functions can call other functions, which can call other functions . . .

```
#include <stdio.h>

int add2(int a, int b) {
    return a + b;
}

int add3(int a, int b, int c) {
    return add2(a, add2(b, c));
}

int main() {
    printf("%d\n", add3(10, 20, 5));
    return 0;
}

$ gcc function_eg5.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
```

35

Functions

A *recursive* function is a function that calls itself

```
// Euclidean algorithm for greatest common divisor
int gcd(int a, int b) {
    if(a == 0) {
        return b;
    }
    return gcd(b % a, a);
}
```

Be careful; it's easy to accidentally write a recursive function that never exits, causing a *stack overflow*

Functions

Saw we have a program that calculates compound interest that is structured like this:

```
float compound(float p, float r, int n) {  
    ...  
}  
  
int main() {  
    ...  
    float monthly = compound(p, r, 12);  
    ...  
}
```

compound is above main. What if we put it below?

Functions

```
#include <stdio.h>
#include <math.h>

int main() {
    printf("%f\n", compound(1000.0, 0.05, 12));
    return 0;
}

float compound(float p, float r, int n) {
    return p * pow(1 + r/n, n);
}
```

Functions

```
$ gcc function_eg6.c -std=c99 -pedantic -Wall -Wextra -lm
function_eg6.c: In function 'main':
function_eg6.c:5:20: warning: implicit declaration of function
  'compound' [-Wimplicit-function-declaration]

    printf("%f\n", compound(1000.0, 0.05, 12));
               ^~~~~~
function_eg6.c:5:14: warning: format '%f' expects argument of
  type 'double', but argument 2 has type 'int' [-Wformat=]

    printf("%f\n", compound(1000.0, 0.05, 12));
               ~~^ ~~~~~
               %d
function_eg6.c: At top level:
function_eg6.c:9:7: error: conflicting types for 'compound'
float compound(float p, float r, int n) {
               ^~~~~~
function_eg6.c:5:20: note: previous implicit declaration of
  'compound' was here

    printf("%f\n", compound(1000.0, 0.05, 12));
               ^~~~~~
```

Functions

If the compiler finds a call to a function that hasn't been defined or declared, it tries to *infer* the parameter and return types

- If it infers correctly, you may only see a warning like the first one (which you should fix anyway)
- If it infers incorrectly, you'll see *more* warnings and errors; above, compiler incorrectly infers return type int

Functions

Compiler needs to know about the function before we call it

Callee must be defined or declared (*prototyped*) before caller

implicit declaration of function warning means this hasn't been done properly; you should fix it, regardless of what compiler says after

Functions

This is a function *definition*

```
int add2(int a, int b) {  
    return a + b;  
}
```

Specifies name, return type, parameters (type and name for each), and implementation

Functions

If implementation is omitted, function is *declared* but not defined.
This is a *function prototype*.

```
int add2(int a, int b);
```

Definition can be given later

When writing function A, you can call function B as long as B has been defined or prototyped *previously*

Functions

```
#include <stdio.h>
#include <math.h>

// function declaration (prototype)
float compound(float p, float r, int n);

int main() {
    printf("%f\n", compound(1000.0, 0.05, 12));
    return 0;
}

// function definition
float compound(float p, float r, int n) {
    return p * pow(1 + r/n, n);
}

$ gcc function_eg7.c -std=c99 -pedantic -Wall -Wextra -lm
$ ./a.out
1051.162598
```

Functions

Parameter names can be omitted in prototypes

```
float compound(float p, float r, int n); // OK
```

```
float compound(float, float, int); // also OK
```

Functions

Functions cannot be “nested” – unlike Python

```
// This WON'T compile
int add3(int a, int b, int c) {
    int add2(int d, int e) {
        return d + e;
    }
    return add2(a, add2(b, c));
}
```

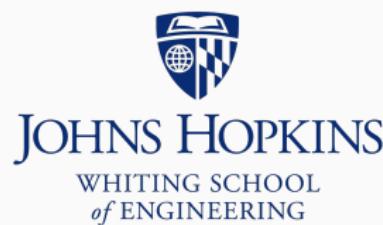
C++ *lambda functions* accomplish something like this; but not discussed here

Program structure

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Program structure

Big C projects are split across many .c source files

Header (.h) are part of the “glue” for combining .c files

Program structure

main.c

```
#include <stdio.h>

int main() {
    float total = compound(100.0, 0.10, 10);
    printf("%0.2f\n", total);
    return 0;
}
```

interest.c

```
#include <math.h>

float compound(float p, float r, int n) {
    return p * pow(1 + r/n, n);
}
```

main calls compound; compound is defined in a different .c file

Two problems:

- main needs to be preceded by a prototype for compound
- main.c and interest.c need to be compiled *together* somehow

Headers

main.c

```
#include <stdio.h>
#include "interest.h"

int main() {
    float total = compound(100.0, 0.10, 10);
    printf("%0.2f\n", total);
    return 0;
}
```

interest.c

```
#include <math.h>

float compound(float p, float r, int n) {
    return p * pow(1 + r/n, n);
}
```

interest.h

```
float compound(float, float, int);
```

Header (.h) files contain prototypes & declarations allowing code in one .c file to use functions & variables in another

interest.h has a prototype for compound

- `#include "interest.h"` allows us to use it

Headers

When #include'ing a .h *you created*, use " " instead of < >

```
#include <stdio.h>    // provided by C
#include <assert.h>    // provided by C
#include "interest.h" // I wrote this
```

Headers

Compile the two .c files together by simply specifying both as arguments to gcc:

```
$ gcc main.c interest.c -std=c99 -pedantic -Wall -Wextra -lm  
$ ./a.out  
110.46
```

You don't have to tell gcc about the .h files; the #include statements take care of that

Headers

What if we forget to #include "interest.h"?

```
$ gcc main_noinc.c interest.c -std=c99 -pedantic -Wall -Wextra -lm
main_noinc.c: In function 'main':
main_noinc.c:4:19: warning: implicit declaration of function 'compound'
[-Wimplicit-function-declaration]
    float total = compound(100.0, 0.10, 10);
                           ^~~~~~
$ ./a.out
0.00
```

A compiler warning and a wrong answer; what happened?

- Hint: look back at notes on functions

Steps of compilation

We discussed compilation steps when we first saw “Hello, World!”

1. Preprocessor

- Gather relevant source code
- Handle #include's and #define's

2. Compiler

- Gather preprocessed code and compile to object code
- If using -c, stop here and output .o files

3. Linker

- Gather *object code* into a single executable file

Steps of compilation

Steps 1/2 and 3 can be separate

```
# Steps 1 & 2 -- preprocess and compile to separate .o's
gcc main.c -c -std=c99 -pedantic -Wall -Wextra -lm
gcc interest.c -c -std=c99 -pedantic -Wall -Wextra -lm

# Step 3 -- combine .o's into single executable
gcc -o main main.o interest.o
```

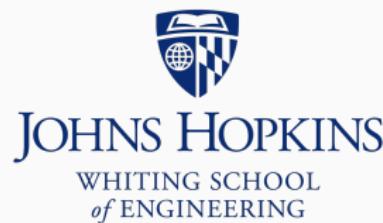
Or they can all happen at once

```
# Steps 1, 2 & 3
gcc -o main main.c interest.c -c -std=c99 -pedantic -Wall -Wextra -lm
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Makefiles

For programs spread across many .c & .h files and executables, make and Makefiles tie things together

Makefile expresses *dependence relationships* between files

Defines *rules* for building one file from others

Makefiles

Example that performs 3 steps (preprocess, compile, link) at once:

- Some gcc arguments omitted for conciseness

```
interest-example: interest.c interest.h main.c  
    gcc -o interest-example interest.c main.c -lm
```

First line says `interest-example` depends on `interest.c`,
`interest.h` and `main.c`

Second line gives command for building `interest-example`

Makefiles

```
$ make interest-example  
gcc -o interest-example interest.c main.c -lm  
$ ./interest-example  
110.46
```

Makefiles

This version performs steps 1 & 2 (preprocess and compile) separately from step 3 (link)

```
interest-example: interest.o main.o  
        gcc -o interest-example interest.o main.o -lm
```

```
interest.o: interest.c interest.h  
        gcc -c interest.c
```

```
main.o: main.c interest.h  
        gcc -c main.c
```

Makefiles

```
$ make interest-example
gcc -c interest.c
gcc -c main.c
gcc -o interest-example interest.o main.o -lm
$ ./interest-example
110.46
```

Makefiles

Makefiles can get repetitive, e.g. if we re-write the gcc arguments for every rule. Use *variables* to simplify:

```
CC = gcc
```

```
CFLAGS = -std=c99 -Wall -Wextra -pedantic
```

```
interest-example: interest.c interest.h main.c
```

```
$(CC) -o interest-example interest.c main.c $(CFLAGS) -lm
```

`$(CC)` is replaced by `gcc`

`$(CFLAGS)` is replaced by `-std=c99 -Wall -Wextra -pedantic`

Vocabulary

```
interest-example: interest.c interest.h main.c  
$(CC) -o interest-example interest.c main.c $(CFLAGS) -lm
```

interest-example is the *target*

interest.c interest.h main.c are the *prerequisites*

\$(CC) -o interest-example ... is the *command*

All these together make up a *rule*

Makefiles

```
interest-example: interest.c interest.h main.c  
    $(CC) $(CFLAGS) -o interest-example interest.c main.c
```

make interest-example asks make to build interest-example target, **if...**

- interest-example doesn't exist, *or*
- interest.c has changed more recently than interest-example, *or*
- interest.h has changed more recently than interest-example, *or*
- main.c has changed more recently than interest-example

If *any* are true, corresponding command is run

Makefiles

make looks at the “whole picture” of how targets are interrelated when deciding what to build

If you want to build target A **and**

- A depends on B *and*
- B depends on C *and*
- B is out of date (i.e. C is newer than B)

. . . then make builds B first, then builds A

Makefiles

For make to work, you must be in the same directory with the Makefile

- Advanced: use `make -C` if in a different directory

Makefile has to be called Makefile

- Advanced: use `make -f <name>` if it's not called Makefile

Typing `make` without specifying a target builds the *default target*; whichever appears first in the Makefile

Makefiles

A *very* common mistake is to use spaces instead of tabs

```
CC = gcc
CFLAGS = -std=c99 -Wall -Wextra -pedantic -lm

interest-example: interest.c interest.h main.c
    $(CC) $(CFLAGS) -o interest-example interest.c main.c
```

You can't tell by looking, but I put 4 spaces instead of a tab before
the \$(CC) command

Makefiles

```
$ make interest-example  
Makefile:5: *** missing separator. Stop.
```

“missing separator” usually means you used spaces instead of tab
emacs *should* notice you’re editing a Makefile and use tabs where appropriate; you can force emacs to use tab with Ctrl-q <tab>

Makefile tutorials

mrbook.org/blog/tutorials/make/

- Uses g++/.cpp instead of gcc/.c, but ideas are the same

www.cs.bu.edu/teaching/cpp/writing-makefiles/

- Uses g++/.cpp instead of gcc/.c, but ideas are the same

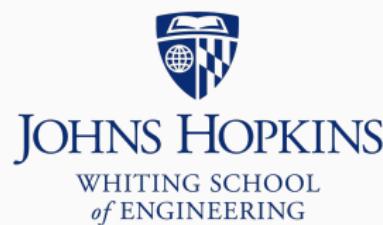
cslibrary.stanford.edu/107/UnixProgrammingTools.pdf

- Section 2

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org

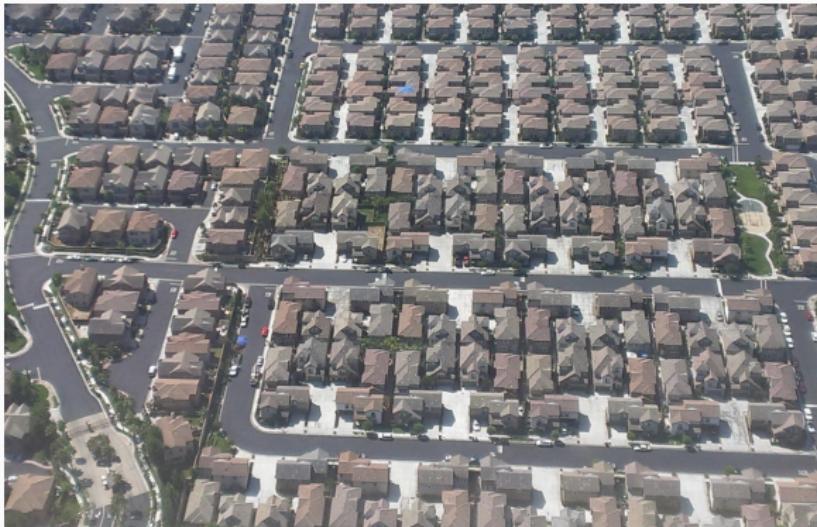


Source markdown available at github.com/BenLangmead/c-cpp-notes

Memory

Variables live in memory

Each variable has an “address” in memory, like a house address



commons.wikimedia.org/wiki/File:South-Los-Angeles-subdivision-houses-near-Darby-Park-Aerial-view-from-north-August-2014.jpg

Memory

Or like a post-office box:



Memory

In this example, fahrenheit and celsius are variables that live at addresses in memory

```
#include <stdio.h>

int main() {
    int fahrenheit = 71;
    float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    printf("%0.2f", celsius);
    return 0;
}
```

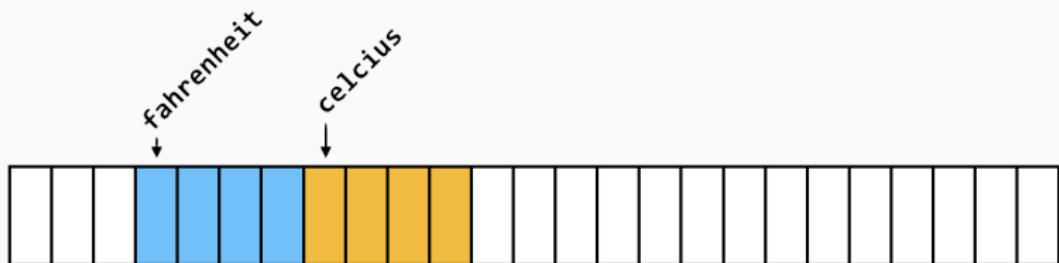
Memory

Memory is like a large array of bytes, like `char[]`

- An *address* is an *offset* into the array
- When a variable occupies >1 byte, address points to *first* byte

Memory

int and float both take 4 bytes, so we might picture them in memory like this:



We'll see diagrams like this frequently

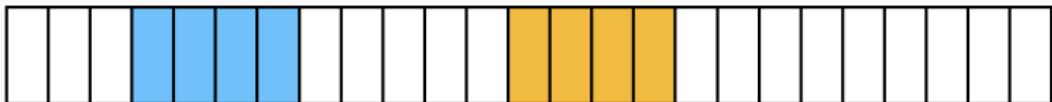
Imagine the leftmost slot's address is 0, the next is 1, etc

In this figure, `fahrenheit` is at address 3 and `celsius` is at 7

Memory

We can't generally predict what addresses our variables will be at

E.g. any of these would have been possible:



Memory

Putting & before a variable name *takes its address*

- We can print an address with printf:

```
#include <stdio.h>

int main() {
    int fahrenheit = 71;
    printf("fahrenheit lives at %p\n", (void*)&fahrenheit);
    //
    //
    return 0;
}
```

^

address-of

```
$ gcc convert_fc_addr.c -Wall -Wextra -std=c99 -pedantic
$ ./a.out
fahrenheit lives at 0x7ffff24c142c
```

Memory

Two unfamiliar things here:

- The `(void*)` before `&fahrenheit`; we will talk more about this when we cover pointers (soon)
- The address itself: `0x7ffd56b72dcc`

`0x7ffd56b72dcc` is just a number!

- `0x` at beginning indicates it's base-16, or *hexadecimal*
- In base-16, digits 0 – 9 aren't enough so use a – f as well
- This number is 140,726,058,298,828 in decimal

Pointers

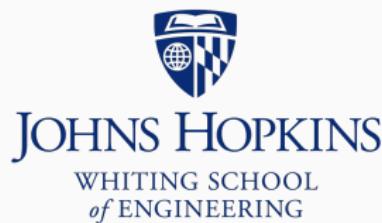


xkcd.com/138/

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Pointers

All problems in computer science can be solved by another level of indirection... .

Pointers

... except of course for the problem of too many indirections.

– David Wheeler

Pointers

A pointer is a variable that holds *the address of (pointer to)* a value

- `int *counter_ptr` is a *pointer to* an integer
- `char *welcome_message` is a *pointer to* a character

Pointers

Pointer can be assigned *the address of* a non-pointer (using &)

```
int counter = 0;           // regular variable  
int *counter_ptr = &counter; // pointer variable
```

counter_ptr gets the address of counter

Pointers

& adds a layer of *indirection*

- &a is the address of (pointer to) a

* removes a layer of indirection

- b is a pointer, *b is the variable it points to

```
int counter = 7;           // variable
int *counter_ptr = &counter; // = counter's address
int counter_copy = *counter_ptr; // = copy of counter
```

Pointers

We can represent code like this:

```
int counter = 7;           // variable  
int *counter_ptr = &counter; // = counter's address  
int counter_copy = *counter_ptr; // = copy of counter
```

By drawing a diagram like this:



Pointers

For normal variables, we write their value in a box labeled with their name

For pointers, we draw an arrow to the variable pointed to

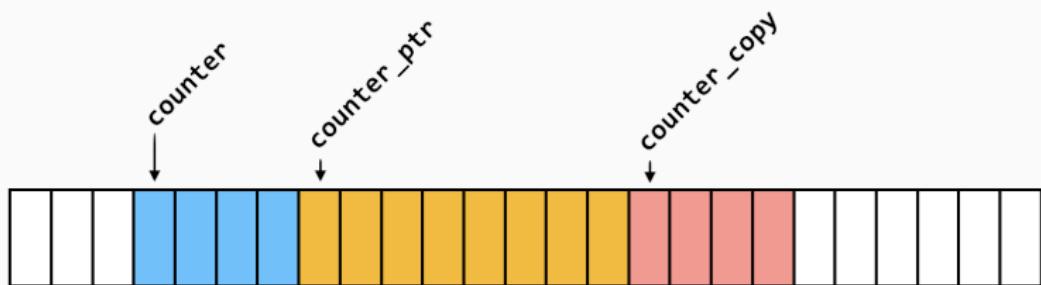


Pointers

Pointers live at addresses in memory just like normal variables

On modern (64-bit) computers, pointers occupy 8 bytes each

Memory layout for previous program might look like this:



Pointers

```
#include <stdio.h>

int main() {
    int a = 40;
    int b = *&a; // reference-then-dereference
    printf("%d\n", b);
    return 0;
}
```

```
$ gcc -c ptr_eg0.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg0 ptr_eg0.o
$ ./ptr_eg0
40
```

Pointers

A dereferenced pointer is something *you can assign to*

- Sometimes called an *lvalue*

```
#include <stdio.h>

int main() {
    int counter = 7;                  // variable
    int *counter_ptr = &counter; // = counter's address
    *counter_ptr = 10;                // this is OK!
    printf("counter=%d\n", counter);
    return 0;
}
```

```
$ gcc -o ptr_eg1 ptr_eg1.c -std=c99 -pedantic -Wall -Wextra
$ ./ptr_eg1
counter=10
```

Pointers

Pointers enable “pass by pointer”, allowing us to modify variables in caller

Pointers

```
#include <stdio.h>

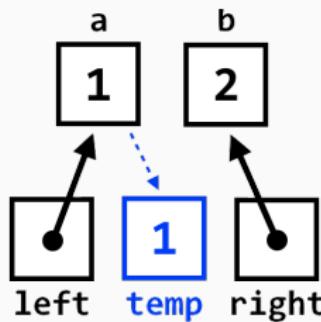
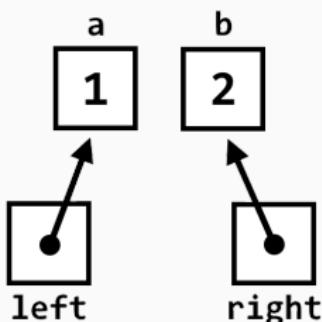
void swap(int *left, int *right) {
    int tmp = *left;
    *left = *right;
    *right = tmp;
}

int main() {
    int a = 1, b = 2;
    swap(&a, &b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}

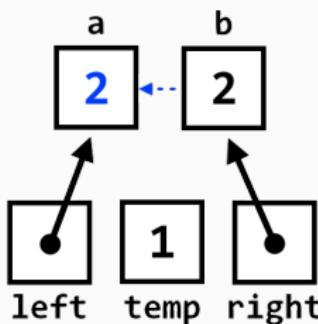
$ gcc -o swap1 swap1.c -std=c99 -pedantic -Wall -Wextra
$ ./swap1
a=2, b=1
```

Pointers

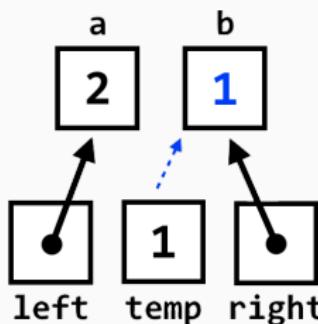
When in doubt, draw a diagram:



`int tmp = *left;`



`*left = *right;`



`*right = tmp;`

Pointers

What's wrong here?

```
#include <stdio.h>

void swap(int left, int right) {
    int tmp = left;
    left = right;
    right = tmp;
}

int main() {
    int a = 1, b = 2;
    swap(a, b);
    printf("a=%d, b=%d\n", a, b);
    return 0;
}
```

Pointers

```
$ gcc -c swap2.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o swap2 swap2.o  
$ ./swap2  
a=1, b=2
```

Forgot to make left and right be int *

Forgot to pass by pointer: swap(&a, &b)

Pointers

A value of NULL indicates an “empty” / “invalid” pointer

So what happens here?

```
char *null_ptr = NULL;  
printf("address %p = %s\n", (void*)null_ptr, null_ptr);
```

Pointers

```
#include <stdio.h>

int main() {
    char *null_ptr = NULL;
    printf("address %p = %s\n", (void*)null_ptr, null_ptr);
    return 0;
}
```

```
$ gcc -c ptr_null_eg2.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_null_eg2 ptr_null_eg2.o
$ ./ptr_null_eg2
address (nil) = (null)
```

Passing NULL as %s argument yields undefined behavior. printf is nice enough to print (null) and not crash, but we can't count on such forgiveness.

Pointers

What happens?

```
char *null_ptr = NULL;  
printf("address %p = %c\n", (void*)null_ptr, *null_ptr);  
//
```



Pointers

```
#include <stdio.h>

int main() {
    char *null_ptr = NULL;
    printf("address %p = %c\n", (void*)null_ptr, *null_ptr);
    //                                     ^ ^
    //                                     ^^^^^^
    return 0;
}
```

This straight up crashes

```
$ gcc -o ptr_null_eg3 ptr_null_eg3.c -std=c99 -pedantic -Wall -Wextra
$ ./ptr_null_eg3
Segmentation fault
```

Pointers

Dereferencing a pointer to memory that doesn't "belong" to you usually results in a segmentation fault or other crash

Dereferencing NULL is a particularly common mistake

- Always check return values for NULL errors (e.g. fopen)

Pointers

```
int a = 7;  
int *p = &a;  
(*p)++;
```

Does this modify p, a, or both?

Pointers

```
#include <stdio.h>

int main() {
    int a = 7;
    int *p = &a;
    printf("%p %d\n", (void*)p, a);
    (*p)++;
    printf("%p %d\n", (void*)p, a);
}
```

```
$ gcc -c ptr_eg4.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg4 ptr_eg4.o
$ ./ptr_eg4
0x7ffdee08f6b4 7
0x7ffdee08f6b4 8
```

Answer: modifies a

Pointers

```
int a = 7;  
int *p = &a;  
p++; // used to be (*p)++
```

Does this modify p, a, or both?

Pointers

```
#include <stdio.h>

int main() {
    int a = 7;
    int *p = &a;
    printf("%p %d\n", (void*)p, a);
    p++;
    printf("%p %d\n", (void*)p, a);
}

$ gcc -c ptr_eg5.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg5 ptr_eg5.o
$ ./ptr_eg5
0x7fff4e188844 7
0x7fff4e188848 7
```

Answer: modifies p

Pointers

Pointer *arithmetic* is possible

- Pointer can be operand in addition and subtraction, causing pointer to “seek” forward and backward across memory slots

```
$ ./ptr_eg5  
0x7fffb585f094 7  
0x7fffb585f098 7
```

Here the pointer advanced by 4 bytes

- It's an `int *` and `int` is 4 bytes long
- We advanced by 1 slot (`p++`), so that's 4 bytes

Pointers

`ptr1 = ptr2` - assignment between same-type pointers works

`ptr1 == ptr2` - true if `ptr1` and `ptr2` point to same place

`ptr1 == NULL` - asks if `ptr1` is `NULL` (equals 0)

Pointers

- You can print a pointer, e.g. `printf("%p", ptr)`
- A pointer is just an (unsigned) integer
- `NULL` equals 0; usually indicates “empty” or “invalid” pointer
- Dereferencing a pointer to memory that doesn’t “belong” to you will result in a crash
- Pointers can be operated on by `+`, `-`, `+=`, `++`, `=`, `==`

Pointers

We saw that these differ in terms of whether p or a is changed:

```
int a = 7;  
int *p = &a;  
p++; // changes p
```

```
int a = 7;  
int *p = &a;  
(*p)++; // changes a
```

Related question: what does it mean for a pointer to be const?

Pointers

Putting `const` before the pointer type means the variable *pointed to* can't be modified

```
#include <stdio.h>
```

```
int main() {
    int a = 7;
    const int *p = &a;
    printf("%p %d\n", (void*)p, a);
    (*p)++;
    printf("%p %d\n", (void*)p, a);
}
```

```
$ gcc -c ptr_const_eg1.c -std=c99 -pedantic -Wall -Wextra
ptr_const_eg1.c: In function 'main':
ptr_const_eg1.c:7:9: error: increment of read-only location '*p'
    (*p)++;
           ^~
```

Pointers

`const int *p` - variable *pointed to* can't be modified

`int * const p` - the *pointer* can't be modified

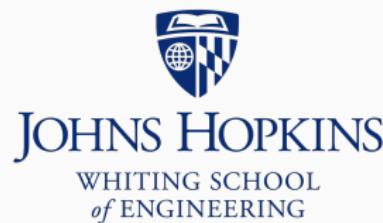
`const int * const p` - *neither* can be modified

Pointers & arrays

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Pointers & arrays

Pointers and arrays are closely related

Say we have array `int a[]`.

- `a[0]` and `*a` are equivalent
- `[...]` is a combination of dereferencing and pointer addition
 - `*(a + 3)` is a synonym for `a[3]`
 - `(a + 3)` is a synonym for `&a[3]`

Pointers & arrays

You'll notice the differences between arrays and pointers when using `sizeof`

```
#include <stdio.h>
int main() {
    int a[] = {0, 1, 2, 3, 4, 5};
    int *a_ptr = a;
    printf("sizeof(a)=%d, sizeof(a_ptr)=%d\n",
           (int)sizeof(a), (int)sizeof(a_ptr));
    return 0;
}
```

```
$ gcc -c ptr_sizeof_eg1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_sizeof_eg1 ptr_sizeof_eg1.o
$ ./ptr_sizeof_eg1
sizeof(a)=24, sizeof(a_ptr)=8
```

Pointers & arrays

Passing array as argument *converts it to a pointer*, losing any information about how long it is

- Sometimes called “array decaying”

Pointers & arrays

```
#include <stdio.h>

void f1(int arg[10]) { printf("f1: %lu\n", sizeof(arg)); }
void f2(int arg[])   { printf("f2: %lu\n", sizeof(arg)); }
void f3(int *arg)    { printf("f3: %lu\n", sizeof(arg)); }

int main() {
    int one_thru_ten[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printf("main: %lu\n", sizeof(one_thru_ten));
    f1(one_thru_ten);
    f2(one_thru_ten);
    f3(one_thru_ten);
    return 0;
}
```

Pointers & arrays

```
$ gcc -o decay1 decay1.c -std=c99 -pedantic -Wall -Wextra
decay1.c: In function 'f1':
decay1.c:3:50: warning: 'sizeof' on array function parameter 'arg' will return
size of 'int *' [-Wsizeof-array-argument]
void f1(int arg[10]) { printf("f1: %lu\n", sizeof(arg)); }
^
decay1.c:3:13: note: declared here
void f1(int arg[10]) { printf("f1: %lu\n", sizeof(arg)); }
^~~

decay1.c: In function 'f2':
decay1.c:4:50: warning: 'sizeof' on array function parameter 'arg' will return
size of 'int *' [-Wsizeof-array-argument]
void f2(int arg[]) { printf("f2: %lu\n", sizeof(arg)); }
^
decay1.c:4:13: note: declared here
void f2(int arg[]) { printf("f2: %lu\n", sizeof(arg)); }
^~~

$ ./decay1
main: 40
f1: 8
f2: 8
f3: 8
```

Compiler warns you

Pointers & arrays

This fits with what we know

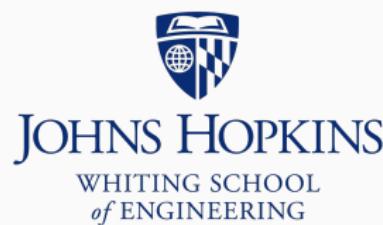
- Passing an array is “pass by pointer,” since arrays decay into pointers when passed
- This is also why we can modify an array in the callee and see the changes in the caller

Lifetime & scope

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Lifetime & scope

Variable's *lifetime* is the period when it exists in memory

- When lifetime ends, memory is reclaimed and can be reused for other variables

Variable's *scope* is the part of the program where you can use it

- Lifetime and scope are *often* the same *but not always*

Lifetime & scope

Scope and lifetime of a variable declared in a block {...} ends at terminal brace }

```
if(a == 7) {  
    int c = 70;  
    printf("%d\n", c);  
}
```

When program reaches }, c is both *out of scope* (we can't refer to it anymore) and *dead* (memory reclaimed)

Lifetime & scope

For for loop index variable, scope is the loop body

```
for(int i = 0; i < 10; i++) {  
    sum += i;  
}  
// after }, i is "dead" and "out of scope"
```

Lifetime & scope

Variables in scope at the time of a function call are *not* in scope in the callee

```
#include <stdio.h>

void print_a() {
    // COMPILER ERROR; can't refer to 'a' here
    printf("a=%d\n", a);
}

int main() {
    int a = 1; // 'a' declared here
    print_a();
    return 0;
}
```

Lifetime & scope

Pointers give a way around this:

```
#include <stdio.h>

void print_a(int *a) {
    printf("*a=%d\n", *a); // OK
}

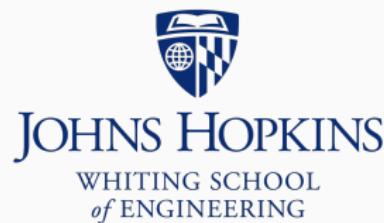
int main() {
    int a = 1; // a declared here
    print_a(&a);
    return 0;
}

$ gcc -o scope2 scope2.c -std=c99 -pedantic -Wall -Wextra
$ ./scope2
*a=1
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Stack & heap

We think of scope and lifetime in the context of *the stack* (or the *call stack*), which grows upwards as:

- New local variables are declared
- Functions call other functions

The bottom of the call stack is always `main` and its local variables

Stack

compound.c

```
0: float compound(float p, float r, int n) {  
1:     return p * pow(1 + r/n, n);  
2: }  
3:  
4: int main() {  
5:     float total = compound(100.0, 0.10, 10);  
6:     printf("%0.2f\n", total);  
7:     return 0;  
8: }
```

Call stack at line 5 (start of main):

main

float total

Call stack at line 1 (in compound):

compound

float p, float r, int n

main

Bookmark: resume at line 5
float total

Upon function call, caller saves a “bookmark” for where to return to when callee finishes. Then room is made on the stack for the callee and its variables.

Stack

When functions return or when scopes are exited, stack shrinks

Stack overflow is when the stack grows so large it exhausts available memory

- E.g. because of a recursive function that never returns

Stack

Explains why a function can't return a locally-declared array:

```
scale.c

0: double* scale(double arr[5], double factor) {
1:     double scaled_array[5];
2:     for(int i = 0; i < 5; i++) {
3:         scaled_array[i] = arr[i] * factor;
4:     }
5:     return scaled_array;
6: }
7:
8: int main() {
9:     double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
10:    double* scaled_array = scale(array, 2.0);
11:    printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
12:    return 0;
13: }
```

1

Call stack at line 10, before call:

main

```
double *scaled_array  
double array[5]
```

2

Call stack at line 5:

```
scale  
double scaled_array[5]  
double factor  
double arr[5]
```

1

3

Call stack at line 10, after call:

main

```
double *scaled_array
    ^^^^^^^^^^^^
    points to scaled_array[5] in scale
    function, but it's dead & reclaimed
double array[5]
```

Stack arrays

When we declare an array, its size must be a “compile-time constant”

```
int array[400]; // we can do this
```

```
#define ARRAY_SIZE 400
```

```
int array[ARRAY_SIZE]; // this is also fine
```

`#define X Y` just means that everytime X appears in the program, it should be replaced with Y. It's a “macro” rather than a variable because the substitution happens in the “preprocessing” step, prior to compilation.

Stack arrays

```
int n = get_length_of_array();
int array[n]; // we shouldn't do this
```

C99 lets you do this, but earlier versions of C don't

It's considered bad style because it's easy to accidentally overflow the stack

- This is the only time you'll see it in these slides

Dynamic memory allocation

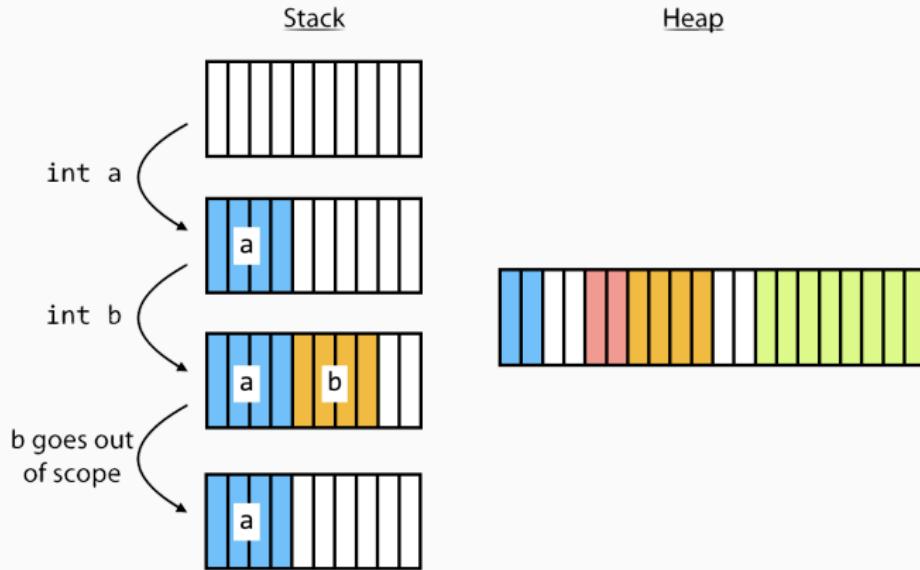
We're about to discuss dynamic memory allocation, where many of these issues are addressed:

- Flexible lifetime; we decide when allocated memory is allocated and deallocated (reclaimed)
- Allocated memory is *not* on the stack, can't cause stack overflow
- Allocation size need not be known at compile time
 - Can be a function of variables in the program

Stack vs. heap

So far, our variables and functions have used *the stack* to store data

We will soon be using a different area called *the heap*



Stack vs. heap

Stack: We declare variables; lifetime same as scope

- C takes care of allocating/deallocating memory as variables enter/exit scope

Heap: Lifetime is under our control

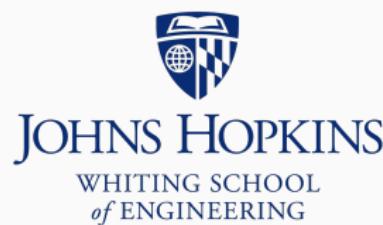
- We explicitly allocate and deallocate
 - E.g. with malloc and free, discussed later
- Operating System is places variables in memory in a non-overlapping way

Dynamic memory allocation

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Dynamic memory allocation

`malloc(size)` allocates `size` bytes and returns a pointer to the allocated memory

The memory is *uninitialized*

`malloc` returns `NULL` upon error (e.g. not enough memory)

Dynamic memory allocation

```
int n = get_length_of_array();
int *array = malloc( ? );
```

Dynamic memory allocation

```
int n = get_length_of_array();
int *array = malloc(n * sizeof(int));
```

`sizeof(int)` is better than simply putting 4; not every compiler/computer will agree that an `int` is 4 bytes

Dynamic memory allocation

Note the types in these prototypes (from man malloc)

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- void *: “wildcard” or “generic” pointer to any type
- size_t: usually the same thing as a long unsigned int

Dynamic memory allocation

Memory allocated with `malloc`, `calloc` or `realloc` is called *dynamically allocated* memory

Each returns a pointer to the newly-allocated memory

Allocated memory must eventually be deallocated, otherwise it “leaks” and isn’t deallocated until the program exits

If your program accumulates enough “leaked” memory, it will exhaust memory and crash

- Possibly taking other programs with it

Dynamic memory allocation

- `void *` is the “wildcard” pointer; can point to any type

```
int *array = malloc(40 * sizeof(int));  
// behind the scenes, C "automatically" changes  
// the void * returned by malloc into an int *
```

```
char *array2 = malloc(40 * sizeof(char));  
// same thing here, but for char *
```

```
float *array3 = malloc(40 * sizeof(float));  
// same thing here, but for float *
```

Dynamic memory allocation

`malloc` allocates memory that is uninitialized at first

`calloc` allocates memory with all positions initialized to 0

`realloc` resizes a previously-allocated chunk of memory

- Useful for dynamically resizing a data structure that grows as more data comes in

`free` deallocates memory returned by any of the above

Dynamic memory allocation

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

double* scale(double arr[5], double factor) {
    double *scaled_array = malloc(sizeof(double) * 5);
    assert(scaled_array != NULL); // bad practice; use if instead
    for(int i = 0; i < 5; i++) {
        scaled_array[i] = arr[i] * factor;
    }
    return scaled_array;
}

int main() {
    double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
    double* scaled_array = scale(array, 2.0);
    printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
    free(scaled_array);
    return 0;
}
```

Dynamic memory allocation

```
$ gcc -c scale_dynamic.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o scale_dynamic scale_dynamic.o  
$ ./scale_dynamic  
2.00 16.60
```

Dynamic memory allocation

This program has a *memory leak*:

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

double* scale(double arr[5], double factor) {
    double *scaled_array = malloc(sizeof(double) * 5);
    assert(scaled_array != NULL); // bad practice; use if instead
    for(int i = 0; i < 5; i++) {
        scaled_array[i] = arr[i] * factor;
    }
    return scaled_array;
}

int main() {
    double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
    double* scaled_array = scale(array, 2.0);
    printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
    // free(scaled_array); // *** LEAK ***
    return 0;
}
```

Dynamic memory allocation

This program has an especially bad leak (repeated in a loop):

```
int super_leaky(int n) {
    int done = 0;
    // assume we iterate many times before setting done=1
    while(!done) {
        int *array = malloc(n * sizeof(int));
        assert(array != NULL);

        // do stuff with array but don't deallocate it

        // we leak more memory *in each iteration*
    }
    return 0;
}
```

Dynamic memory allocation

Is this OK?

```
int possibly_leaky(int n) {  
    int array[10];  
    // do stuff with array  
    return 0;  
}
```

Dynamic memory allocation

Yes, this is perfectly OK

`int array[10]` is on the stack so allocation and deallocation are handled for us, behind the scenes

We're not attempting to return array, which would be bad

```
int possibly_leaky(int n) { // not leaky
    int array[10];
    // do stuff with array
    return 0;
}
```

Dynamic memory allocation

Be a good memory citizen

- Always explicitly deallocate memory you've allocated
- When you add an allocation to your program, think about where you should deallocate

Dynamic memory allocation

What should we have done differently?

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

int *sequence(int n) {
    int *x = malloc(n * sizeof(int));
    assert(x != NULL);
    for(int i = 0; i < n; i++) {
        x[i] = i;
    }
    return x;
}

int main() {
    int *seq = sequence(10);
    for(int i = 0; i < 10; i++) {
        printf("%d ", seq[i]);
    }
    return 0;
}
```

Dynamic memory allocation

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

// Returns a newly-allocated array
int *sequence(int n) {
    int *x = malloc(n * sizeof(int));
    assert(x != NULL);
    for(int i = 0; i < n; i++) {
        x[i] = i;
    }
    return x;
}

int main() {
    int *seq = sequence(10);
    // seq is a newly allocated array
    // you remind yourself "I have to deallocate this!"
    for(int i = 0; i < 10; i++) {
        printf("%d ", seq[i]);
    }
    free(seq); // that's better!!!
    return 0;
}
```

Dynamic memory allocation

Only free memory you allocated with malloc & friends

Don't free the same thing twice

free the same *exact* pointer you allocated; this won't work:

```
int *array = malloc(20 * sizeof(int));
assert(array != NULL);
array++; // skip over first element
free(array); // no longer equals pointer returned by malloc
```

Dynamic memory allocation

calloc is like malloc except:

The allocated memory is first initialized to all 0s

1st parameter to calloc is number of elements

2nd parameter is element size

```
int *array = malloc(40 * sizeof(int));
for(int i = 0; i < 40; i++) {
    array[i] = 0;
}
```

// ...is the same as...

```
int *array = calloc(40, sizeof(int));
```

Dynamic memory allocation

realloc “resizes” an existing memory allocation

Dynamic memory allocation

```
#include <stdio.h>
#include <stdlib.h> // for malloc, free & friends

int main() {
    int *array = malloc(10 * sizeof(int));
    for(int i = 0; i < 10; i++) {
        array[i] = i * 2;
    }
    array = realloc(array, 20 * sizeof(int));

    // elements 0 through 9 are still set as above!
    // only need to set 10 through 19 here
    for(int i = 10; i < 20; i++) {
        array[i] = i * 2;
    }

    for(int i = 0; i < 20; i++) {
        printf("%d ", array[i]);
    }
    putchar('\n'); // print single newline to stdout
    return 0;
}

$ gcc -c realloc_eg1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o realloc_eg1 realloc_eg1.o
$ ./realloc_eg1
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Dynamic memory allocation

realloc's first parameter must have been allocated previously with malloc (or calloc or realloc)

```
int array[] = {0, 2, 4, 6};  
realloc(array, 8 * sizeof(int)); // won't work!
```

Dynamic memory allocation

```
new = realloc(old, new_size)
```

- If new size is greater than old size, the new space is uninitialized

Behind the scenes, realloc will:

- new = malloc(new_size)
- Copy contents of old into beginning of new
- free(old)

Dynamic memory allocation

Be careful: after calling realloc, don't use any pointers corresponding to older versions of that allocation. The old memory was deallocated and no longer belongs to you.

Dynamic memory allocation

A quick reminder as to how big everything is

```
#include <stdio.h>

int main() {
    printf("char: %d\n",           (int)sizeof(char));
    printf("int: %d\n",            (int)sizeof(int));
    printf("unsigned int: %d\n",   (int)sizeof(unsigned int));
    printf("float: %d\n",          (int)sizeof(float));
    printf("double: %d\n",         (int)sizeof(double));
    printf("void *: %d\n",         (int)sizeof(void *));
    printf("int *: %d\n",          (int)sizeof(int *));
    printf("size_t: %d\n",         (int)sizeof(size_t));
}
```

Dynamic memory allocation

```
$ gcc -c sizeof_all.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o sizeof_all sizeof_all.o  
$ ./sizeof_all  
char: 1  
int: 4  
unsigned int: 4  
float: 4  
double: 8  
void *: 8  
int *: 8  
size_t: 8
```

Dynamic memory allocation

New powers:

- Can (finally) allocate arbitrary-length arrays
- Arrays/variables allocated in this way live across scopes and function calls, until we deallocate

Dynamic memory allocation

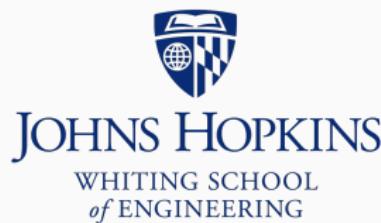
New responsibilities:

- For every allocation, we must deallocate (no leaks)
- Check return value from `malloc` & friends; could be `NULL`
- Don't dereference an address (pointer) that doesn't point to your own properly-allocated memory
- Any new memory allocated by `malloc` or `realloc` is uninitialized; assign to it before using it
- Don't forget that `free` and `realloc` deallocate their arguments

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Valgrind



Image from valgrind.org

Very easy-to-use tool for finding memory leaks and other pointer/memory mistakes

Compile your program with -g option for more helpful output from valgrind

```
valgrind --leak-check=full ./your-program <arg1> <arg2> ...
```

valgrind

From valgrind.org/docs/manual/faq.html:

The “grind” is pronounced with a short ‘i’ – ie. “grinned” (rhymes with “tinned”) rather than “grined” (rhymes with “find”). Don’t feel bad: almost everyone gets it wrong at first.

Valgrind is the name of the main entrance to Valhalla (the Hall of the Chosen Slain in Asgard).

valgrind

```
#include <stdio.h>

int main() {
    printf("  ***  My program's output  ***\n");
    return 0;
}
```

valgrind

```
$ gcc -o valgrind_eg1 valgrind_eg1.c -std=c99 -pedantic -Wall -Wextra -g
$ valgrind --leak-check=full ./valgrind_eg1
*** My program's output ***
==22== Memcheck, a memory error detector
==22== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==22== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22== Command: ./valgrind_eg1
==22==
==22==
==22== HEAP SUMMARY:
==22==     in use at exit: 0 bytes in 0 blocks
==22==   total heap usage: 1 allocs, 1 frees, 4,096 bytes allocated
==22==
==22== All heap blocks were freed -- no leaks are possible
==22==
==22== For counts of detected and suppressed errors, rerun with: -v
==22== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

valgrind

Output of the program is interspersed with messages from valgrind

Some valgrind messages have to do with invalid reads and writes

- Usually, instances where we've dereferenced addresses not “belonging” to us

Everything from HEAP SUMMARY on has to do with memory leaks

- Failing to deallocate a pointer you allocated earlier

valgrind

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

char *string_copy(const char *orig) {
    char *fresh = malloc(strlen(orig) * sizeof(char));
    assert(fresh != NULL);
    strcpy(fresh, orig);
    return fresh;
}

int main() {
    char *hello_copy = string_copy("hello");
    assert(hello_copy != NULL);
    printf("%s\n", hello_copy);
    return 0;
}
```

valgrind

valgrind output indicates two problems:

“Invalid write” and “invalid read”

- We dereferenced addresses that didn’t belong to us

valgrind

```
$ gcc -o buggy_strcpy buggy_strcpy.c -std=c99 -pedantic -Wall -Wextra -g
$ valgrind --leak-check=full ./buggy_strcpy
==21== Memcheck, a memory error detector
==21== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==21== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==21== Command: ./buggy_strcpy
==21==
==21== Invalid write of size 1
==21==   at 0x4C32C9D: strcpy (vg_replace_strmem.c:510)
==21==   by 0x40065D: string_copy (buggy_strcpy.c:9)
==21==   by 0x400675: main (buggy_strcpy.c:14)
==21== Address 0x5221045 is 0 bytes after a block of size 5 alloc'd
==21==   at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
==21==   by 0x400626: string_copy (buggy_strcpy.c:7)
==21==   by 0x400675: main (buggy_strcpy.c:14)
==21==
==21== Invalid read of size 1
==21==   at 0x4C32B94: strlen (vg_replace_strmem.c:458)
==21==   by 0x4EB4D41: puts (in /usr/lib64/libc-2.26.so)
==21==   by 0x4006A5: main (buggy_strcpy.c:16)
==21== Address 0x5221045 is 0 bytes after a block of size 5 alloc'd
==21==   at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
==21==   by 0x400626: string_copy (buggy_strcpy.c:7)
==21==   by 0x400675: main (buggy_strcpy.c:14)
```

valgrind

```
==21== HEAP SUMMARY:
==21==     in use at exit: 5 bytes in 1 blocks
==21==   total heap usage: 2 allocs, 1 frees, 4,101 bytes allocated
==21==
==21== 5 bytes in 1 blocks are definitely lost in loss record 1 of 1
==21==    at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
==21==    by 0x400626: string_copy (buggy_strcpy.c:7)
==21==    by 0x400675: main (buggy_strcpy.c:14)
==21==
==21== LEAK SUMMARY:
==21==    definitely lost: 5 bytes in 1 blocks
==21==    indirectly lost: 0 bytes in 0 blocks
==21==    possibly lost: 0 bytes in 0 blocks
==21==    still reachable: 0 bytes in 0 blocks
==21==          suppressed: 0 bytes in 0 blocks
==21==
==21== For counts of detected and suppressed errors, rerun with: -v
==21== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

valgrind

Let's start with the “stack trace” for the memory leak:

```
--21== 5 bytes in 1 blocks are definitely lost in loss record 1 of 1
--21==    at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
--21==    by 0x400626: string_copy (buggy_strcpy.c:7)
--21==    by 0x400675: main (buggy_strcpy.c:14)
```

Look for the topmost function that's actually part of the code you wrote, and go to the file and line number indicated.

We wrote `main` & `string_copy`, but not `malloc`. `string_copy` is highest, so go to `buggy_strcpy.c:7`:

```
char *fresh = malloc(strlen(orig) * sizeof(char));
```

valgrind

valgrind is saying that we fail to free the memory returned by this malloc

That's true! We should free it in main:

```
int main() {
    char *hello_copy = string_copy("hello");
    assert(hello_copy != NULL);
    printf("%s\n", hello_copy);
    free(hello_copy); // that's better
    return 0;
}
```

valgrind

```
==21== Invalid write of size 1
==21==   at 0x4C32C9D: strcpy (vg_replace_strmem.c:510)
==21==   by 0x40065D: string_copy (buggy_strcpy.c:9)
==21==   by 0x400675: main (buggy_strcpy.c:14)
==21== Address 0x5221045 is 0 bytes after a block of size 5 alloc'd
==21==   at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
==21==   by 0x400626: string_copy (buggy_strcpy.c:7)
==21==   by 0x400675: main (buggy_strcpy.c:14)
```

Warning has two parts:

- Top stack trace: where “invalid write” happened
- Bottom: Where a nearby memory block was allocated; useful since mistake is usually that we go past the end of an allocated block

valgrind

```
char *string_copy(const char *orig) {  
    // *** memory allocated on next line ***  
    char *fresh = malloc(strlen(orig) * sizeof(char));  
    assert(fresh != NULL);  
    // *** invalid write on next line ***  
    strcpy(fresh, orig);  
    return fresh;  
}
```

What's the mistake?

valgrind

`strlen` returns length of string *not counting* null terminator

But we need to `malloc` enough chars for string *and* terminator

valgrind

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

char *string_copy(const char *orig) {
    char *fresh = malloc((strlen(orig)+1) * sizeof(char)); // ** FIX 2
    assert(fresh != NULL);
    strcpy(fresh, orig);
    return fresh;
}

int main() {
    char *hello_copy = string_copy("hello");
    assert(hello_copy != NULL);
    printf("%s\n", hello_copy);
    free(hello_copy); // ** FIX 1
    return 0;
}
```

valgrind

Now let's look at the invalid read:

```
==21== Invalid read of size 1
==21==      at 0x4C32B94: strlen (vg_replace_strmem.c:458)
==21==      by 0x4EB4D41: puts (in /usr/lib64/libc-2.26.so)
==21==      by 0x4006A5: main (buggy_strcpy.c:16)
==21== Address 0x5221045 is 0 bytes after a block of size 5 alloc'd
==21==      at 0x4C2FB6B: malloc (vg_replace_malloc.c:299)
==21==      by 0x400626: string_copy (buggy_strcpy.c:7)
==21==      by 0x400675: main (buggy_strcpy.c:14)
```

This is because the lack of null terminator causes the call to `printf` (which the compiler turned into a call to `puts`) to read beyond the end of `hello_copy`. We already fixed this.

valgrind

After fixes, we have a clean valgrind report:

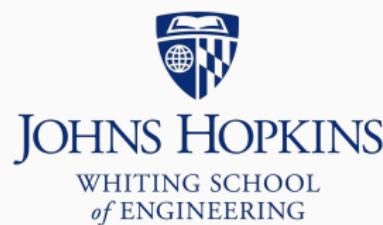
```
$ gcc -o fixed_strcpy fixed_strcpy.c -std=c99 -pedantic -Wall -Wextra -g
$ valgrind --leak-check=full ./fixed_strcpy
hello
==34== Memcheck, a memory error detector
==34== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==34== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==34== Command: ./fixed_strcpy
==34==
==34==
==34== HEAP SUMMARY:
==34==     in use at exit: 0 bytes in 0 blocks
==34==   total heap usage: 2 allocs, 2 frees, 4,102 bytes allocated
==34==
==34== All heap blocks were freed -- no leaks are possible
==34==
==34== For counts of detected and suppressed errors, rerun with: -v
==34== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Debugging with gdb

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Debugging with gdb

gdb: GNU debugger

gdb helps you run your program in a way that allows you to:

- flexibly *pause* and *resume*
- print out the values of variables mid-stream
- see where severe errors like Segmentation Faults happen

When using gdb (or valgrind) we compile with -g, which packages up the source code (“debug symbols”) along with the executable

gdb

Keep this “cheat sheet” nearby as you read on:

bit.ly/gdb_cheat

Buggy program:

```
#include <stdio.h>
#include <string.h>

void string_reverse(char *str) {
    const int len = strlen(str);
    for(int i = 0; i < len; i++) {
        str[i] = str[len-i-1]; // swap characters
        str[len-i-1] = str[i];
    }
}

int main() {
    char reverse_me[] = "AAABBB";
    string_reverse(reverse_me);
    printf("%s\n", reverse_me);
    return 0;
}
```

gdb

We are trying to reverse a string by starting at the left and right extremes, swapping the characters, then continuing inward, swapping as we go until we've reversed the whole thing.

```
$ gcc -o str_rev str_rev.c -std=c99 -pedantic -Wall -Wextra -g  
$ ./str_rev  
BBBBBB
```

Oops, I expected output to be BBBAAA

valgrind gives clean report, so likely not an issue with mishandled pointers

gdb

We'll use gdb to investigate

Since the problem would seem to be in the `string_reverse` function, I am going to start my program at the beginning and then take small steps forward until I get to the loop.

gdb

```
(gdb) break main
Breakpoint 1 at 0x4005ad: file str_rev.c, line 13.
(gdb) run
Starting program: /app/str_rev
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.26-15.fc27.x86_64

Breakpoint 1, main () at str_rev.c:13
13     char reverse_me[] = "AAABBB";
```

break main because I want to debugger to pause as soon I as get to the beginning of the program, i.e. the main function

run to start the program, which immediately pauses at top of main

After running a command, gdb prints out the next line of code in the program

gdb

```
(gdb) next  
14      string_reverse(reverse_me);  
(gdb) step  
string_reverse (str=0x7fffffff629 "AAABBB") at str_rev.c:5  
5      const int len = strlen(str);
```

next executes the statement on the current line and moves onto the next. If the statement contains a function call, gdb executes it without pausing.

step begins to execute the statement on the current line. If the statement contains a function call, it *steps into* the function and pauses there. Otherwise, it behaves like next.

Now we're at the beginning of string_reverse

gdb

```
(gdb) n  
6      for(int i = 0; i < len; i++) {  
(gdb) print len  
$2 = 6
```

n is short for next

print prints out the value of a variable. len is 6 – that's what we expected. So far so good.

We're about to enter the loop.

gdb

```
(gdb) n  
7      str[i] = str[len-i-1]; // swap characters  
(gdb) p i  
$3 = 0  
(gdb) p str[i]  
$4 = 65 'A'  
(gdb) p str[len-i-1]  
$5 = 66 'B'
```

p is short for print

i's initial value is 0, as expected

The elements we're swapping really are the first A and the last B, as expected

gdb

Let's execute the swap:

```
(gdb) n  
8      str[len-i-1] = str[i];  
(gdb) n  
6      for(int i = 0; i < len; i++) {  
(gdb) p i  
$6 = 0
```

Just finished the first iteration; i still equals 0

gdb

Let's see if the swap was successful:

```
(gdb) p str[i]
$7 = 66 'B'
(gdb) p str[len-i-1]
$8 = 66 'B'
```

No – the swap fails because I overwrite `str[i]` with the value of `str[len-i-1]` *before* copying it into `str[len-i-1]`

This explains why the result is BBBBBB

I need to use a temporary variable like we did previously with swap

Fixed?:

```
#include <stdio.h>
#include <string.h>

void string_reverse(char *str) {
    const int len = strlen(str);
    for(int i = 0; i < len; i++) {
        int temp = str[i]; // swap characters -- FIXED
        str[i] = str[len-i-1];
        str[len-i-1] = temp;
    }
}

int main() {
    char reverse_me[] = "AAABBB";
    string_reverse(reverse_me);
    printf("%s\n", reverse_me);
    return 0;
}
```

gdb

```
$ gcc -o str_rev2 str_rev2.c -std=c99 -pedantic -Wall -Wextra -g  
$ ./str_rev2  
AAABBB
```

Still not working! I expected output to be BBBAAA

Exercise: use gdb to find lingering bug.

Hint 1: examine results of the swaps through *several* loop iterations

Hint 2: Instead of break main, use break str_rev2.c:7, replacing str_rev2.c with the name of your source file and 7 with the line number of the first statement in the loop body. That way run will advance directly there. (If you already set the main breakpoint, remove it with delete.)

gdb help

Type help at the (gdb) prompt for help

- (gdb) help running – for advancing thru program
- (gdb) help show – for printing commands

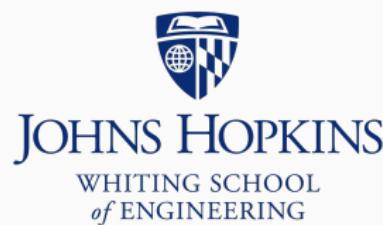
There are *many* gdb commands, so I prefer brief “cheat sheets”:

- darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Structures

A collection of related variables, bundled into one

```
struct card {  
    char rank;  
    char suit;  
};
```

Structures

Variables in a struct are *fields*

```
struct cc_receipt {  
    float amount;  
    char cc_number[16];  
};
```

Two fields: float named amount, and char[16] named number

Structures

Say we're programming a checkers game.

We want a struct describing everything about a game piece

```
struct checkers_piece {  
    // ???  
};
```



Structures

```
struct checkers_piece {  
    int x; // horizontal offset  
    int y; // vertical offset  
    int black; // 0 = white, non-0 = black  
};
```

Structures

```
#include <stdio.h>

struct date {
    int year;
    int month;
    int day;
};

int main() {
    struct date today; // like 3 variables in 1!
    today.year = 2016; // use . to refer to fields
    today.month = 2;
    today.day = 26;
    printf("Today's date: %d/%d/%d\n",
           today.month, today.day, today.year);
    return 0;
}
```

Structures

```
$ gcc -c struct_eg1.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o struct_eg1 struct_eg1.o  
$ ./struct_eg1  
Today's date: 2/26/2016
```

Structures

The struct name { ... }; syntax defines a new struct

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
  
// declare variable 'today' with type 'struct date'  
struct date today;
```

Structures

struct variable can be initialized in similar way to an array:

```
struct date {  
    int year;  
    int month;  
    int day;  
};
```

...

```
struct date today = {2016, 2, 26};
```

Structures

struct fields can be other structs

```
struct date {  
    int year;  
    int month;  
    int day;  
};  
  
struct cc_transaction {  
    // struct within a struct is fine!  
    struct date purchase_date;  
    float amount;  
    char cc_number[16];  
};
```

Structures

struct fields can be pointers

```
struct player {  
    int home_runs;  
    int strikeouts;  
    int walks;  
};  
  
struct team {  
    struct player *catcher;  
    struct player *first_baseman;  
    struct player *second_baseman;  
    ...  
};
```

Structures

`sizeof(struct player)` returns total size of all fields

- With a caveat we'll see later

```
struct date {  
    int year;  
    int month;  
    int day;  
};
```

What is `sizeof(struct date)`?

Structures

```
#include <stdio.h>

struct date {
    int year; // 4 bytes
    int month; // 4 bytes
    int day; // 4 bytes
};

int main() {
    printf("%d\n", (int)sizeof(struct date));
}
```

```
$ gcc -c struct_sizeof.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof struct_sizeof.o
$ ./struct_sizeof
```

12

Structures

A struct can be a function parameter and/or return type

```
struct date next_day(struct date d) {
    if((++d.day) > 30) { // assume 30-day months
        d.day = 1;
        if((++d.month) > 12) {
            d.month = 1;
            d.year++;
        }
    }
    return d;
}
```

Structures

What if it were a void function, without return d at the end?

```
void next_day(struct date d) {  
    if((++d.day) > 30) { // assume 30-day months  
        d.day = 1;  
        if((++d.month) > 12) {  
            d.month = 1;  
            d.year++;  
        }  
    }  
}
```

Structures

structs are passed by value

- next_day on previous slide has no effect
- Alternative 1: return a new struct
- Alternative 2: pass *pointer* to struct

Structures

```
void next_day_in_place(struct date *d) {  
    if((++(*d).day) > 30) {  
        (*d).day = 1;  
        if((++(*d).month) > 12) {  
            (*d).month = 1;  
            (*d).year++;  
        }  
    }  
}
```

Structures

d->day is a synonym for (*d).day

```
void next_day_in_place(struct date *d) {
    if((++d->day) > 30) {
        d->day = 1;
        if((++d->month) > 12) {
            d->month = 1;
            d->year++;
        }
    }
}
```

Structures

```
#include <stdio.h>
#include "date.h" // "struct date" defined here

struct date next_day(struct date d) {
    if(++d.day > 30) {
        d.day = 1;
        if(++d.month > 12) {
            d.month = 1;
            d.year++;
        }
    }
    return d;
}

int main() {
    struct date today = {2016, 2, 26};
    struct date tomorrow = next_day(today);
    printf("Tomorrow's date: %d/%d/%d\n",
           tomorrow.month, tomorrow.day, tomorrow.year);
}

$ gcc -c struct_next_day_1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_next_day_1 struct_next_day_1.o
$ ./struct_next_day_1
Tomorrow's date: 2/27/2016
```

Structures

```
#include <stdio.h>
#include "date.h" // "struct date" defined here

void next_day_in_place(struct date *d) {
    if(++d->day) > 30) {
        d->day = 1;
        if((++d->month) > 12) {
            d->month = 1;
            d->year++;
        }
    }
}

int main() {
    struct date today = {2016, 12, 30};
    next_day_in_place(&today);
    printf("Tomorrow's date: %d/%d/%d\n",
           today.month, today.day, today.year);
    return 0;
}

$ gcc -c struct_next_day_2.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_next_day_2 struct_next_day_2.o
$ ./struct_next_day_2
Tomorrow's date: 1/1/2017
```

Structures

You can have an array of structs

```
struct album {  
    const char *name;  
    const char *artist;  
    double length;  
};  
  
struct album music_collection[99999];  
music_collection[0].name = "The Next Day";  
music_collection[0].artist = "David Bowie";  
music_collection[0].length = 41.9;  
music_collection[1].name = "Hunky Dory";  
...
```

Structures

What is sizeof(struct album)?

```
struct album {  
    const char *name;  
    const char *artist;  
    double length; // 8 bytes  
};
```

Structures

```
$ gcc -c struct_sizeof_album.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o struct_sizeof_album struct_sizeof_album.o  
$ ./struct_sizeof_album  
sizeof(struct album) = 24
```

24 bytes

- const char *s are just (8-byte) pointers
- Strings themselves not stored in the struct

Structures

You can have a struct with an array in it:

```
struct cc_receipt {  
    float amount;  
    char cc_number[16];  
};
```

Structures

What is sizeof(struct cc_receipt)?

```
struct cc_receipt {  
    float amount; // 4 bytes  
    char cc_number[16];  
};
```

Structures

```
$ gcc -c sizeof_receipt.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o sizeof_receipt sizeof_receipt.o  
$ ./sizeof_receipt  
sizeof(struct cc_receipt) = 20
```

Answer: 20 bytes. char cc_number[16] is inside the struct, taking up 16 bytes.

Structures

```
#include <stdio.h>

struct ten_ints {
    int ints[10];
};

void func1(struct ten_ints ints) {
    printf("func1 sizeof(ints)=%d\n", (int)sizeof(ints));
}

void func2(int *ints) {
    printf("func2 sizeof(ints)=%d\n", (int)sizeof(ints));
}

int main() {
    struct ten_ints ints;
    func1(ints);
    func2(ints.ints);
    return 0;
}

$ gcc -c struct_sizeof_receipt.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o struct_sizeof_receipt struct_sizeof_receipt.o
$ ./struct_sizeof_receipt
func1 sizeof(ints)=40
func2 sizeof(ints)=8
```

Structures

When a struct is passed to a function, everything inside is copied, including arrays

This means an array wrapped in a struct is actually passed by value!

- In contrast to normal arrays, which are passed by pointer

Structures

We might get tired of writing struct over and over:

```
struct cc_receipt {  
    float amount;  
    char cc_number[16];  
};  
  
struct cc_receipt lunch_receipt;  
struct cc_receipt dinner_receipt;
```

Structures

We can use `typedef` to make the type name shorter:

```
typedef struct { // no name here
    float amount;
    char cc_number[16];
} cc_receipt;      // name down here

cc_receipt lunch_receipt;
cc_receipt dinner_receipt;
```

Now the type simply `cc_receipt` instead of `struct cc_receipt`

Structures

Size of struct is *at least* the sum of the sizes of its fields

It can be bigger if the compiler decides to add “padding”

```
struct plane {  
    int passengers;  
    double cargo_weight;  
};
```

Structures

```
$ gcc -c sizeof_plane.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o sizeof_plane sizeof_plane.o  
$ ./sizeof_plane  
sizeof(struct plane) = 16
```

For obscure efficiency reasons, the compiler put 4 bytes of “spacer” between the int & double, making total size = 16

Structures

Structures can be defined in a nested way:

```
typedef struct {
    struct {      // this struct type doesn't have a name;
        int r;   // it's just used once to declare a
        int b;   // field named color
        int g;
    } color;
    struct {      // again, no name
        int x;
        int y;
    } position;
} pixel;

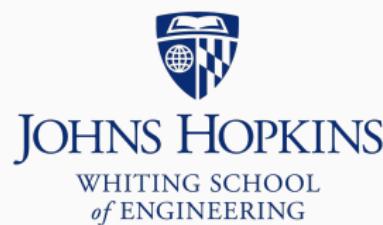
pixel p;
p.color.r = 255;
p.position.x = 40;
p.position.y = 50;
```

Beyond 1D arrays

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Beyond 1D arrays

We've already seen `char* argv[]`: an array of strings

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("argc = %d\n", argc);
    for(int i = 0; i < argc; i++) {
        printf("argv[%d] = %s\n", i, argv[i]);
    }
    return 0;
}
```

```
$ gcc args_eg_1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out rosebud
argc = 2
argv[0] = ./a.out
argv[1] = rosebud
```

Arrays of strings

```
#include <stdio.h>

void print_list(char **message, const int num) {
    for(int i = 0; i < num; i++) {
        printf("%s ", message[i]);
    }
    putchar('\n');
}

int main() {
    char *message1[] = { "Hello", "world!" };
    char *message2[] = { "Have", "a", "nice", "day" };
    print_list(message1, 2);
    print_list(message2, 4);
    return 0;
}
```

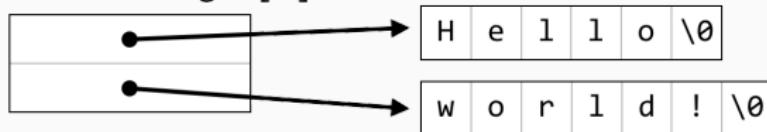
Arrays of strings

```
$ gcc array_of_strs.c -std=c99 -pedantic -Wall -Wextra  
$ ./a.out  
Hello world!  
Have a nice day
```

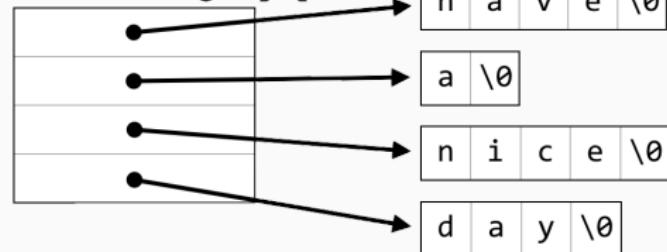
Arrays of strings

```
char *message1[] = { "Hello", "world!" };
char *message2[] = { "Have", "a", "nice", "day" };
```

char *message1[2]



char *message2[4]



Two-dimensional arrays

Say we want to represent a tic tac toe board

O	-	X
<hr/>		
-	O	-
<hr/>		
X	-	X

An array seems appropriate, but we want a *two-dimensional* array

For each element, we'll use a char: ('X', 'O' or '-')

Two-dimensional arrays

Option 1:

```
char board[9];
```

board is a *one-dimensional* array with 9 elements, but we use it to represent a 3x3 board

To access element at row i, column j, use `board[i * 3 + j]`

0	1	2	O	-	X
3	4	5	-	O	-
6	7	8	X	-	X

This is *row-major* order; it is common in practice

Two-dimensional arrays

```
#include <stdio.h>

void print_board(char board[]) {
    for(int i = 0; i < 3; i++) {          // rows
        for(int j = 0; j < 3; j++) {    // columns
            printf("%c ", board[i * 3 + j]);
        }
        putchar('\n');
    }
}

int main() {
    char board[9] = {'O', 'O', 'X',
                    '-', 'O', '-',
                    'X', '-', 'X'};
    print_board(board);
    return 0;
}
```

Two-dimensional arrays

```
$ gcc ttt_ex.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
0 0 X
- 0 -
X - X
```

Two-dimensional arrays

Option 2:

```
char board[3][3];
```

board is a *two-dimensional* array with 3 rows and 3 columns

C “understands” the rows and columns; to access element at row i, column j, use board[i][j]

[0][0]	[0][1]	[0][2]	O	-	X
[1][0]	[1][1]	[1][2]	-	O	-
[2][0]	[2][1]	[2][2]	X	-	X

Two-dimensional arrays

```
#include <stdio.h>

void print_board(char board[][][3]) {
    for(int i = 0; i < 3; i++) {          // rows
        for(int j = 0; j < 3; j++) {    // columns
            printf("%c ", board[i][j]);
        }
        putchar('\n');
    }
}

int main() {
    char board[3][3] = {{'0', '0', 'X'},
                        {'-', '0', '-'},
                        {'X', '-', 'X'}};
    print_board(board);
    return 0;
}
```

Two-dimensional arrays

```
$ gcc ttt_ex_2.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
0 0 X
- 0 -
X - X
```

Two-dimensional arrays

Type of the array parameter is important; we *must* specify the parameter type along with

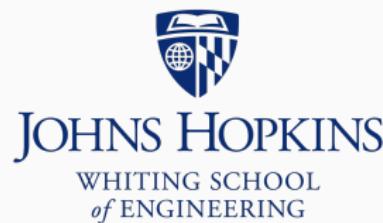
```
void print_board(char board[][][3]) {  
    // ...  
}
```

Binary I/O

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Binary I/O

When reading from a filehandle, we have used fgets and fscanf

These are for *parsing text*, i.e. turning strings like “0.435” and “999” into floats, ints, etc

But you can also read and write data (floats, ints, structs etc) as *binary* data, without turning them into strings first

For midterm project, we will read and write PPM files, using binary I/O

fwrite

```
size_t fwrite(const void *data, size_t size,  
             size_t nitems, FILE *stream);
```

- data: pointer to data to be written
- size: size of 1 item
- nitems: # items to write
- stream: filehandle to write to
 - Must have been opened in "wb" mode

Returns the number of items successfully written

- If return value != nitems, there was an error; check ferror

fread

```
size_t fread(void *data, size_t size,  
            size_t nitems, FILE * stream);
```

- data: data should be copied to here
 - There needs to be enough space! $\text{size} \times \text{nitems}$ bytes
- size: size of 1 item
- nitems: # items to read
- stream: filehandle to read from
 - Must have been opened in "rb" mode

Returns the number of items successfully read

- If return value \neq nitems, there was an error; check ferror

Binary I/O

```
#include <stdio.h> // printf, fread & fwrite
#include <assert.h>

int main() {
    int evens[] = {2, 4, 6, 8}, odds[] = {1, 3, 5, 7};

    FILE *out = fopen("bio_eg1.bin", "wb");
    assert(out != NULL);
    size_t nwritten = 0;
    nwritten += fwrite(evens, sizeof(int), 4, out);
    nwritten += fwrite(odds, sizeof(int), 4, out);
    assert(nwritten == 8);
    fclose(out);

    FILE *in = fopen("bio_eg1.bin", "rb");
    assert(in != NULL);
    int buf1[4] = {0}, buf2[4] = {0};
    size_t nread = 0;
    nread += fread(buf1, sizeof(int), 4, in);
    nread += fread(buf2, sizeof(int), 4, in);
    assert(nread == 8);
    fclose(in);

    printf("Even: %d %d %d %d\n", buf1[0], buf1[1], buf1[2], buf1[3]);
    printf("Odds: %d %d %d %d\n", buf2[0], buf2[1], buf2[2], buf2[3]);
    return 0;
}
```

Binary I/O

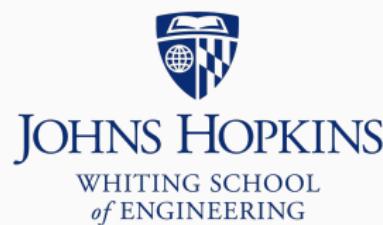
```
$ gcc bio_eg1.c -std=c99 -pedantic -Wall -Wextra  
$ ./a.out  
Even: 2 4 6 8  
Odds: 1 3 5 7
```

Numeric types

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



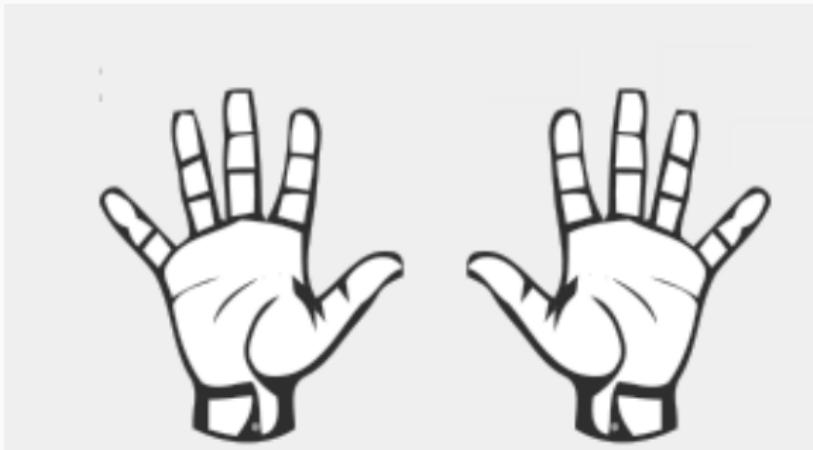
Source markdown available at github.com/BenLangmead/c-cpp-notes

Numeric types

In computers, all data are stored in binary

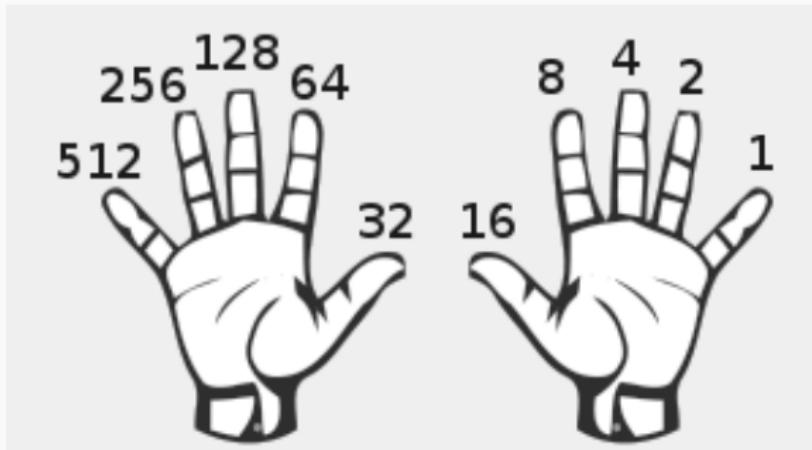
Binary is the number system where each digit is a power of 2

We are used to powers of 10 (decimal)



Bits and binary

If we used our fingers to count in binary, we could count to
 $2^{10} - 1 = 1023$



<https://biscitmx.com/category/unplugged/>

Bits and binary

Integer is like an array of bits, but we can't use [] for individual bits

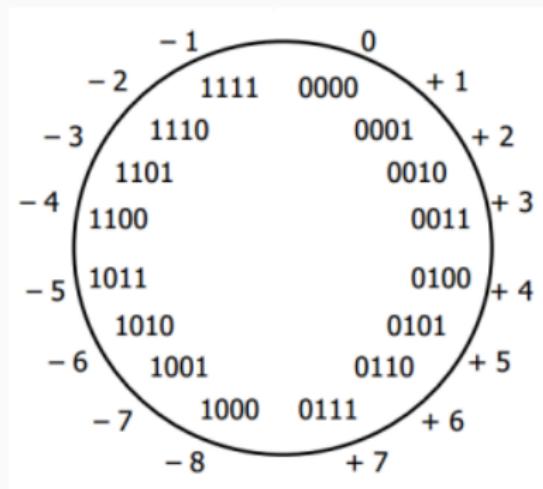
Binary:	0	0	1	1	0	1	0	1
Place value:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53$$

- Instead, we need *bitwise operators*, discussed later

Bits and binary

C integers use “two’s complement” representation for signed integers. Illustration with 4 bits:



http://www.bogotobogo.com/cplusplus/quiz_bit_manipulation.php

When a two’s complement number overflows, it wraps around to a negative number

Bits and binary

```
#include <stdio.h>

int main() {
    int i = 2147483647;
    int i_plus_1 = i + 1;
    printf("i = %d, i+1 = %d\n", i, i_plus_1);
    return 0;
}
```

```
$ gcc -c overflow.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o overflow overflow.o
$ ./overflow
i = 2147483647, i+1 = -2147483648
```

Bits and binary

Floating point numbers use their bits to store a few different things:

- Sign: 1 bit, positive or negative
- Exponent
- Mantissa

sign exponent mantissa

0	1	1	0	1	0	1	1
---	---	---	---	---	---	---	---



$$+ (0.1011)_2 * 10^{(110)_{\text{3bit excess-}k}}$$

$$= (0.1011)_2 * 10^2$$

$$= (10.11)_2$$

$$= (1 * 2)^1 + (1 * 2)^0 + (0 * 2)^{-1} + (1 * 2)^{-2} = 2.75$$

Bits and binary

Integer and floating-point representations differ:

- Integers have limited range, but integers in the range can be represented precisely. Floating point have limited range and can only approximate most numbers in the range.
- Integers use all available bits for two's-complement representation. Floating point have separate sets of bits for sign, exponent and mantissa.

Bits and binary

float a = 1 or int i = 3.0, it's not as simple as copying bits

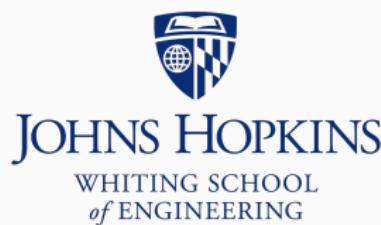
When going from integer types to float (or double), we are getting an approximation, not the exact integer

Casting, promotion and narrowing

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Casting, promotion and narrowing

The type for an *integer literal* (e.g. 88, -1000000000) is determined based on its value

Specifically: the smallest integer type that can store it without overflowing

Casting & numeric types

```
#include <stdio.h>

int main() {
    int a = 1;
    int b = -3000;
    long c = 10000000000; // too big for an int
    printf("%d, %d, %ld\n", a, b, c);
    return 0;
}
```

```
$ gcc -c int_literal.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o int_literal int_literal.o
$ ./int_literal
1, -3000, 10000000000
```

Casting & numeric types

Compiler can warn us when an integer literal is too big

```
#include <stdio.h>

int main() {
    int a = 10000000000; // too big for an int
    // a will "overflow" and "wrap around" to some other number
    printf("%d\n", a);
    return 0;
}

$ gcc -c int_literal2.c -std=c99 -pedantic -Wall -Wextra
int_literal2.c: In function 'main':
int_literal2.c:4:13: warning: overflow in implicit constant conversion
[-Woverflow]
        int a = 10000000000; // too big for an int
                           ^~~~~~
$ gcc -o int_literal2 int_literal2.o
$ ./int_literal2
1410065408
```

Casting & numeric types

Floating-point literal (e.g. 3.14, -.7, -1.1e-12) has type double

You can force it to be float by adding f suffix

```
#include <stdio.h>

int main() {
    float a = 3.14f;
    double b = 33.33, c = -1.1e-12;
    printf("%f, %f, %e\n", a, b, c);
    return 0;
}
```

```
$ gcc -c float_literal.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o float_literal float_literal.o
$ ./float_literal
3.140000, 33.330000, -1.100000e-12
```

Casting & numeric types

Again, compiler will warn if literal doesn't fit

```
#include <stdio.h>

int main() {
    float a = 2e128f; // too big; max float exponent is 127
    double b = 2e1024; // too big; max double exponent is 1023
    printf("%f, %f\n", a, b);
    return 0;
}
```

Casting & numeric types

```
$ gcc -c float_literal2.c -std=c99 -pedantic -Wall -Wextra
float_literal2.c: In function 'main':
float_literal2.c:4:5: warning: floating constant exceeds range of 'float'
[-Woverflow]
    float a =  2e128f; // too big; max float exponent is 127
    ^~~~~
float_literal2.c:5:5: warning: floating constant exceeds range of 'double'
[-Woverflow]
    double b = 2e1024; // too big; max double exponent is 1023
    ^~~~~~
$ gcc -o float_literal2 float_literal2.o
$ ./float_literal2
inf, inf
```

Promotion

C can automatically convert between types “behind the scenes”

This is called *promotion* or *automatic conversion*

```
float ten = 10;  
// float <- int
```

Promotion

```
#include <stdio.h>

int main() {
    int a = 1;
    float f = a * 1.5f;
    printf("%f\n", f);
    return 0;
}
```

```
$ gcc -c promotion_1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o promotion_1 promotion_1.o
$ ./promotion_1
1.500000
```

Promotion

```
int a = 1;  
float f = a * 1.5f;
```

Note operands types: a is int, 1.5f is float

When operand types don't match, “smaller” type is promoted to “larger” before applying operator

char < int < unsigned < long < float < double

Promotion

char < int < unsigned < long < float < double

E.g. $1 + 1.0$: int 1 is converted *directly* to double before addition;
types “in between” (unsigned, long, float) aren’t involved

Promotion

```
#include <stdio.h>

int main() {
    int a = 3;
    float f = a / 2;
    printf("%f\n", f);
    return 0;
}
```

Promotion

```
$ gcc -c promotion_2.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o promotion_2 promotion_2.o  
$ ./promotion_2  
1.000000
```

Promotion

```
int a = 3;  
float f = a / 2;
```

No promotion here, since operands a and 2 are same type: int

Narrowing

Type conversions that are *not* promotions (e.g. double -> float or long -> int) can happen automatically too

Sometimes called *narrowing* conversions

```
#include <stdio.h>

int main() {
    unsigned long a = 1000;
    int b = a; // automatic *narrowing* conversion
    double c = 3.14;
    float d = c; // automatic *narrowing* conversion
    printf("b=%d, d=%f\n", b, d);
    return 0;
}
```

Narrowing

```
$ gcc -c narrow_1.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o narrow_1 narrow_1.o  
$ ./narrow_1  
b=1000, d=3.140000
```

No warnings

Narrowing

```
#include <stdio.h>

int square(int num) {
    return num * num;
}

int main() {
    printf("square(2.5)=%d\n", square(2.5));
    return 0;
}
```

```
$ gcc -c narrow_2.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o narrow_2 narrow_2.o
$ ./narrow_2
square(2.5)=4
```

2.5 becomes 2 when passed to square. No compiler warning.

Narrowing

A value's type is narrowed *automatically* and *without a compiler warning* when: (a) assigning to a variable of narrower type, and (b) passing an argument into a parameter of narrower type.

Other automatic narrowing situations typically yield compiler warnings

Narrowing

```
#include <stdio.h>

int main() {
    printf("sizeof(long)=%d\n", sizeof(long));
    return 0;
}

$ gcc -c casting_1.c -std=c99 -pedantic -Wall -Wextra
casting_1.c: In function 'main':
casting_1.c:4:27: warning: format '%d' expects argument of type 'int', but
argument 2 has type 'long unsigned int' [-Wformat=]
    printf("sizeof(long)=%d\n", sizeof(long));
                           ^
                           %ld

$ gcc -o casting_1 casting_1.o
$ ./casting_1
sizeof(long)=8
```

Casting

Some types just can't be used for certain things. E.g. a float can't be an array index:

```
#include <stdio.h>

int main() {
    int array[] = {2, 4, 6, 8};
    float f = 3.0f;
    printf("array[0]=%d, array[%f]=%d\n", array[0], f, array[f]);
    return 0;
}
```

Casting

```
$ gcc -c casting_2.c -std=c99 -pedantic -Wall -Wextra
casting_2.c: In function 'main':
casting_2.c:6:61: error: array subscript is not an integer
    printf("array[0]=%d, array[%f]=%d\n", array[0], f, array[f]);
                                         ^
                                         ^
```

Casting

Casting gives you control over when promotion or narrowing happen in your program

Casting is sometimes the only way to avoid compiler errors and warnings

Even when conversion would happen automatically, making it explicit with casting can make your code clearer

Casting

```
#include <stdio.h>

int main() {
    int a = 3;
    float f = (float)a / 2;
    //
    // a gets *cast* to float, therefore
    // 2 gets *promoted* to float before division
    printf("%f\n", f);
    return 0;
}
```

```
$ gcc -c casting_3.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o casting_3 casting_3.o
$ ./casting_3
1.500000
```

Casting

```
#include <stdio.h>

int main() {
    printf("sizeof(long)=%d\n", (int)sizeof(long));
    //
    return 0;
}
```

```
$ gcc -c casting_4.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o casting_4 casting_4.o
$ ./casting_4
sizeof(long)=8
```

Casting

```
#include <stdio.h>

int main() {
    int array[] = {2, 4, 6, 8};
    float f = 3.0f;
    printf("array[0]=%d, array[%d]=%d\n",
           array[0], (int)f, array[(int)f]);
    //
    return 0;
}
```

```
$ gcc -c casting_5.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o casting_5 casting_5.o -lm
$ ./casting_5
array[0]=2, array[3]=8
```

Casting

Pointers can get automatically converted too:

```
int *buf = malloc(400 * sizeof(int));  
//      ^^^ void * -> int *  
  
fread(buf, sizeof(int), 400, file);  
//      ^^^ int * -> void *
```

Casting

You can cast pointers. Cast might be necessary to avoid a warning:

```
#include <stdio.h>

int main() {
    int n = 40;
    printf("address %p = %d\n", (void*)&n, n);
    //
    return 0;
}
```

```
$ gcc -c ptr_eg3.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_eg3 ptr_eg3.o
$ ./ptr_eg3
address 0x7ffd11443e2c = 40
```

Casting

Different types are represented differently in memory, so casting from one type of pointer to the other is almost never OK:

```
#include <stdio.h>

int main() {
    int hello[] = {'h', 'e', 'l', 'l', 'o', '\0'};
    printf("%s\n", (char*)hello);
    return 0;
}
```

```
$ gcc -c ptr_cast.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o ptr_cast ptr_cast.o
$ ./ptr_cast
h
```

Type mystery

```
#include <stdio.h>
#include <math.h>

int main() {
    float p = 2000.0, r = 0.10;
    float ci_1 = p * pow(1 + r, 10);
    float ci_2 = p * pow(1.0f + r, 10);
    float ci_3 = p * pow(1.0 + r, 10);
    printf("%.3f\n%.3f\n%.3f\n", ci_1, ci_2, ci_3);
    return 0;
}
```

```
$ gcc -c casting_6.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o casting_6 casting_6.o -lm
$ ./casting_6
5187.486
5187.486
5187.485
```

Type mystery: solved

Prototype for pow: double pow(double, double);

Type promotions are happening both because of the addition and because of the call to pow

- ci_1: 1 converted to float, then added to r, then result is converted to double
- ci_2: 1.0f + r converted to double
- ci_3: r converted to double, then added to 1.0 (already a double)

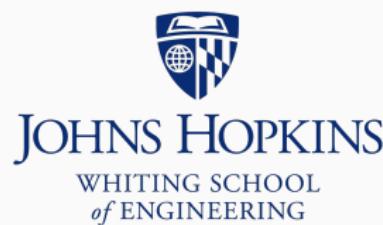
(float)1 and (double)1 are not the same. ci_1 and ci_2 use (float)1, ci_3 uses (double)1.

Bitwise operators

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Bitwise operators

We saw that integers can be used as *boolean* values with $0 = \text{false}$ and $\text{non-}0 = \text{true}$

Also saw logical operators for combining booleans

Operator	Function	Example	Result
<code>&&</code>	Both true (<i>AND</i>)	<code>1 && 0</code>	false (0)
<code> </code>	Either true (<i>OR</i>)	<code>1 0</code>	true (non-0)
<code>!</code>	Opposite (<i>NOT</i>)	<code>!(1 0)</code>	false (0)

Bitwise operators

Saw that integer types consist of bits

Each bit could be considered a boolean true/false value

Binary:	0	0	1	1	0	1	0	1
Place value:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$$2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53$$

Bitwise operators

Bitwise operators performs a function across all bits in integer operands, treating them as boolean true/false values

Bitwise AND (&) performs logical AND (&&) across all bits:

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25

00001100
& 00011001

00001000 = 8 (In decimal)

Bitwise operators

```
#include <stdio.h>
int main() {
    int a = 12;
    int b = 25;
    printf("%d & %d = %d\n", a, b, a & b);
    return 0;
}
```

```
$ gcc bitwise_and.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
12 & 25 = 8
```

Bitwise operators

Since ints are 32-bit (4-byte) values, this is a more accurate picture:

00000000000000000000000000001100 = 12

& 000000000000000000000000000011001 = 25

00000000000000000000000000001000 = 8 (In decimal)

Bitwise operators

Bitwise OR (|) performs logical OR (||):

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

00001100	
00011001	
<hr/>	
00011101	= 29 (In decimal)

Bitwise operators

```
#include <stdio.h>
int main() {
    int a = 12;
    int b = 25;
    printf("%d | %d = %d\n", a, b, a | b);
    return 0;
}
```

```
$ gcc bitwise_or.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
12 | 25 = 29
```

Bitwise operators

$x \ll n$ shifts bits of x the left N positions

N 0s are “shifted in” at right-hand side

N bits “fall off” left-hand side

$25 = 00011001$ (In Binary)

Bitwise left-shift of 25 by 5 positions ($25 \ll 5$)

11001

$\ll 5$

1100100000 = 800 (In decimal)

Bitwise operators

```
#include <stdio.h>
int main() {
    int a = 25;
    int b = 5;
    printf("%d << %d = %d\n", a, b, a << b);
    return 0;
}
```

```
$ gcc bitwise_lshift.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
25 << 5 = 800
```

Bitwise operators

Similar for bitwise right shift (`>>`)

`25 = 00011001` (In Binary)

Bitwise right-shift of 25 by 4 positions (`25 >> 4`)

`00011001`

`>> 4`

`00000001 = 1`

Bitwise operators

```
#include <stdio.h>
int main() {
    int a = 25;
    int b = 4;
    printf("%d >> %d = %d\n", a, b, a >> b);
    return 0;
}
```

```
$ gcc bitwise_rshift.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
25 >> 4 = 1
```

Bitwise operators

```
#include <stdio.h>
int main() {
    int num = 53;
    char bin_str[33] = {'\0'};
    int tmp = num;
    for(int i = 0; i < 32; i++) {
        if((tmp & 1) != 0) {      // least significant bit set?
            bin_str[31-i] = '1'; // prepend 1
        } else {
            bin_str[31-i] = '0'; // prepend 0
        }
        tmp >>= 1;             // shift right by 1
    }
    printf("%d in binary: %s\n", num, bin_str);
    return 0;
}
```

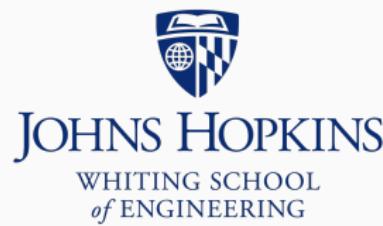
Bitwise operators

```
$ gcc bitwise_convert.c -std=c99 -pedantic -Wall -Wextra  
$ ./a.out  
53 in binary: 0000000000000000000000000000110101
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

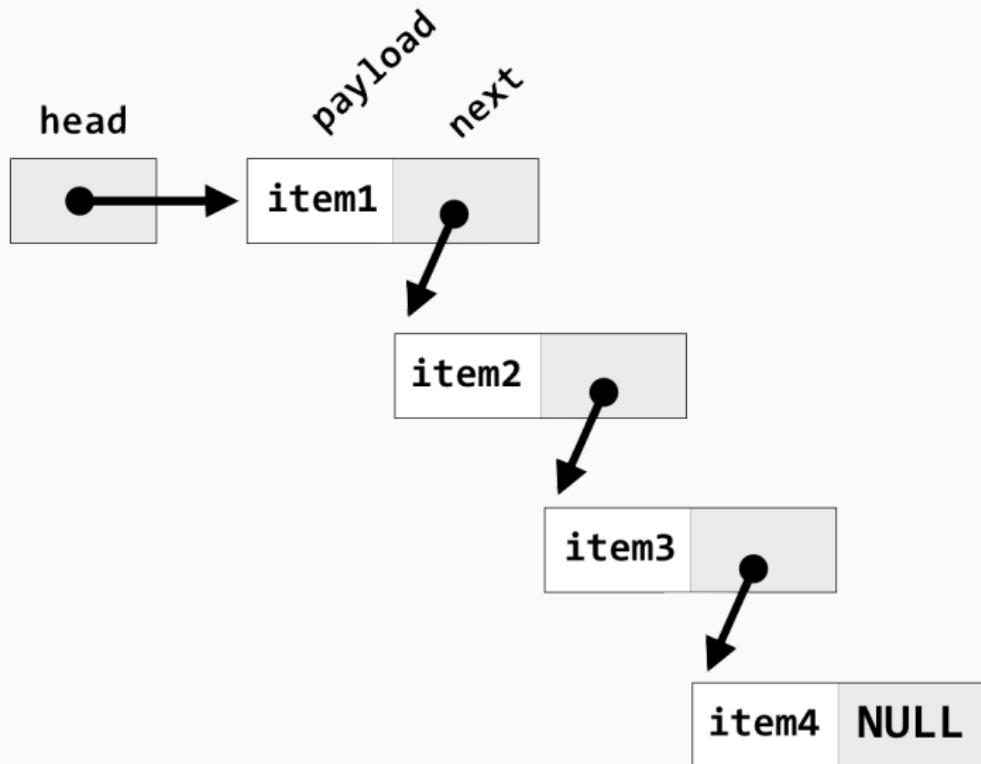
Linked lists

Sequence of structures (*nodes*) connected by pointers (*links*)

Each node has relevant data associated (*payload*)

- music_collection might have an album payload
- calendar might have a date payload

Linked lists



Linked lists

A linked-list node will be a struct

Two components:

- Payload
- Pointer to next node

```
struct Node {  
    // ?  
};
```

Linked lists

```
struct Album {  
    const char *name;  
    const char *artist;  
    double length;  
};  
  
typedef struct _Node {  
    struct Album payload;  
    struct _Node *next;  
} Node;
```

Linked lists

We will use this definition for Node:

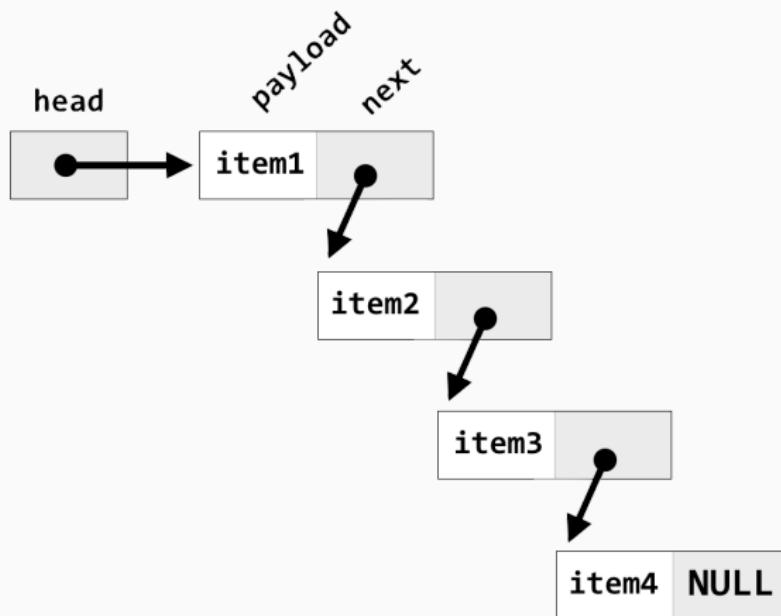
```
typedef struct _Node {  
    struct Album payload;  
    struct _Node *next;  
} Node;
```

Has both a long name (`struct _Node`) and a `typedef`'ed short name (`Node`)

- Usually, we can simply use `Node` (thanks to `typedef`)
- When declaring the type for the next pointer, `typedef` isn't active yet, so we have to use `struct _Node`

Linked lists

Apart from the nodes, a *head pointer* to the first (*head*) node

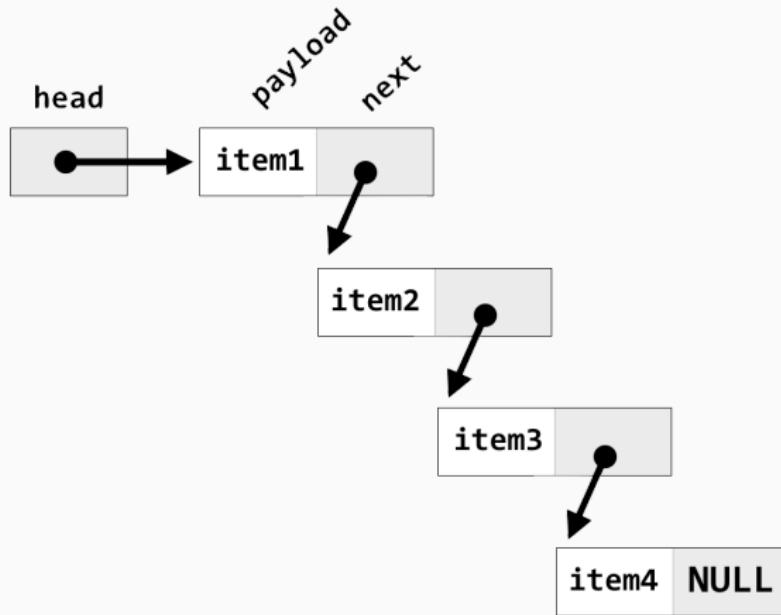


What is the head pointer's type?

Linked lists

Answer: Node *

Linked lists



How do we know when we've reached the end (*tail*) of the list?

Linked lists

Answer: tail's next pointer should equal NULL

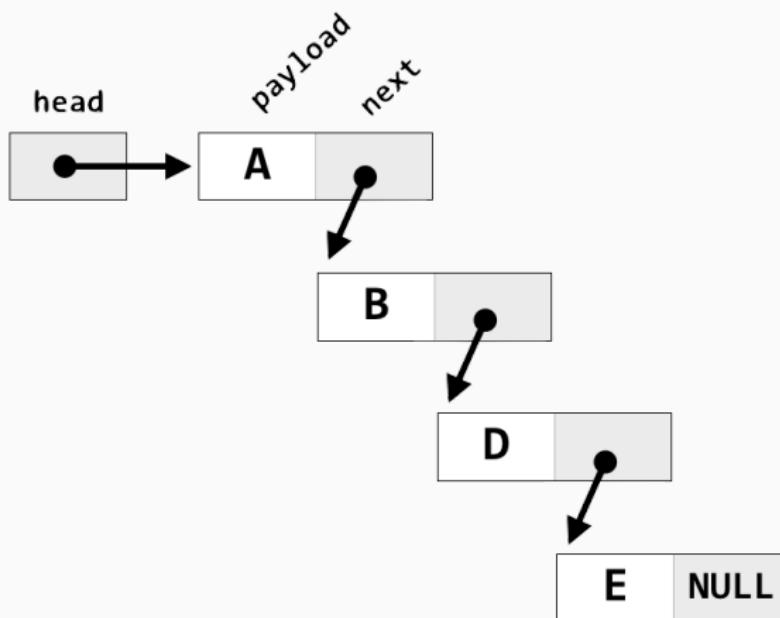
Linked lists

We'll write functions for three basic linked-list tasks:

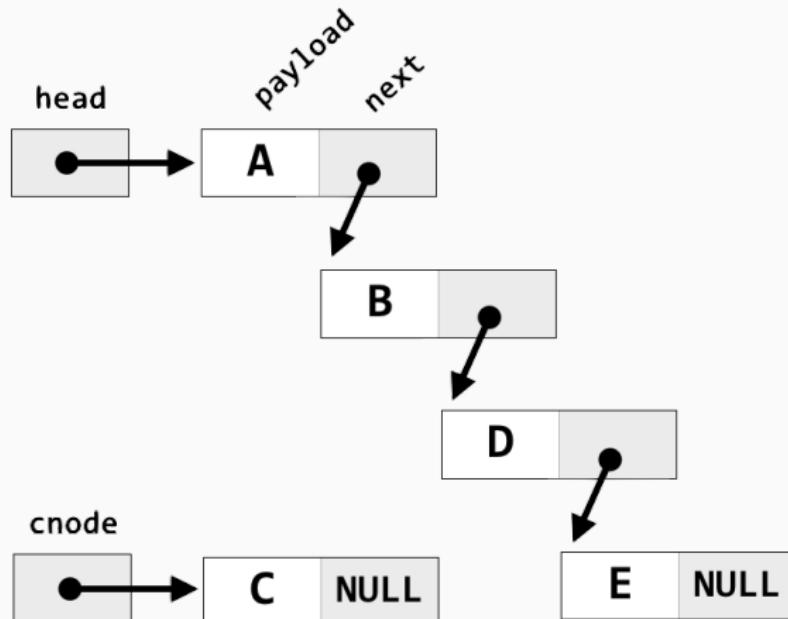
- Inserting an item
- Deleting an item
- “Walking along” (“traversing”) the list

Linked lists: inserting

How to insert a new node?

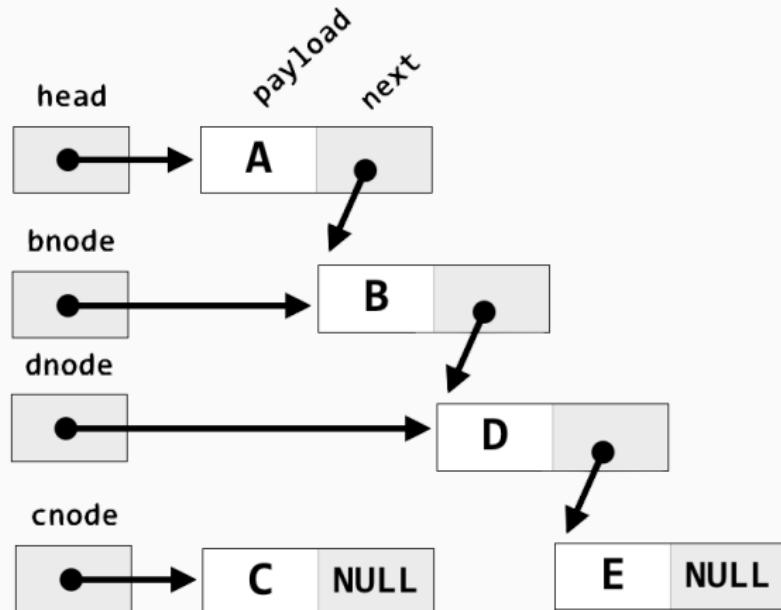


Linked lists: inserting



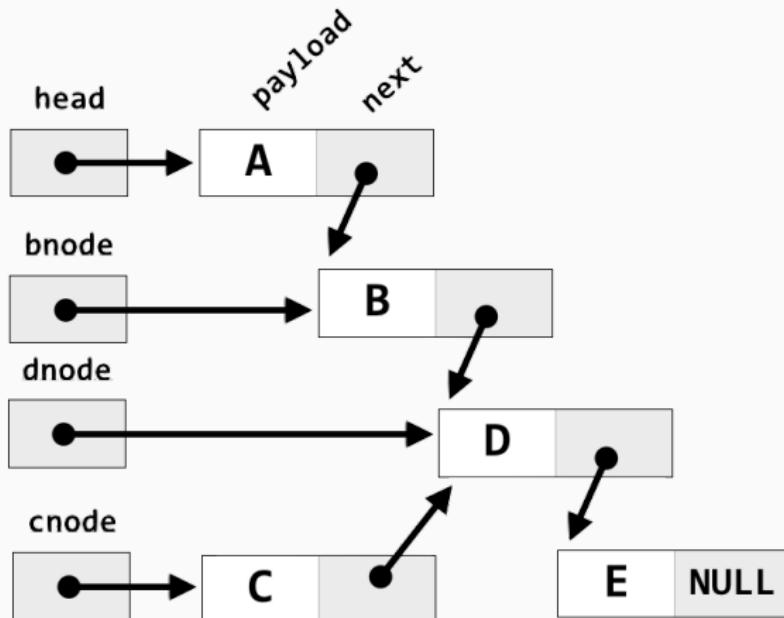
```
Node *cnode = malloc(sizeof(Node));  
cnode->payload = 'C';  
cnode->next = NULL;
```

Linked lists: inserting



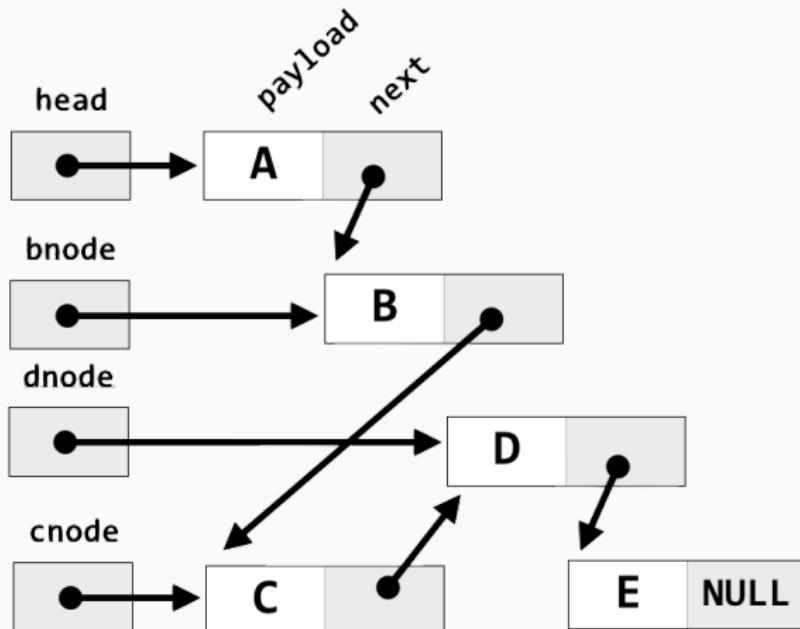
```
Node *bnode = head->next;  
Node *dnode = bnode->next;
```

Linked lists: inserting



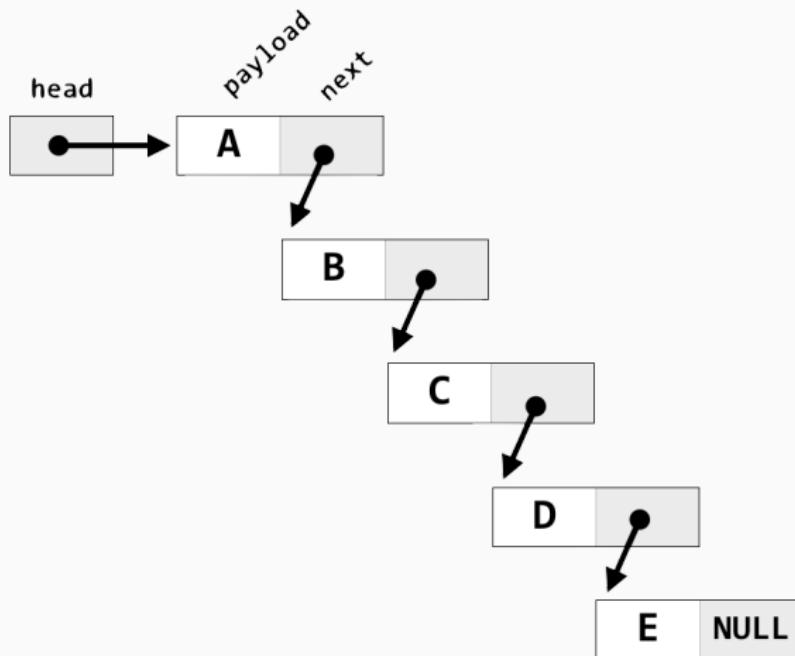
```
cnode->next = dnode; // make new node point to successor
```

Linked lists: inserting



```
bnode->next = cnode; // make predecessor point to new node
```

Linked lists: inserting



Linked lists: inserting

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct _Node {
    char payload;
    struct _Node *next;
} Node;

int insert_after(Node *before, char newPayload) {
    Node *newNode = malloc(sizeof(Node));
    if(newNode == NULL) {
        return -1; // error
    }
    newNode->payload = newPayload;
    newNode->next = before->next; // make new node point to successor
    before->next = newNode; // make predecessor point to new node
    return 0;
}
```

Linked lists: inserting

```
// *Traverses* list, as we discuss later
void print_list(Node *cur) {
    while(cur != NULL) {
        printf("%c ", cur->payload);
        cur = cur->next;
    }
    putchar('\n');
}

int main() {
    // Make a little list
    Node *head = malloc(sizeof(Node));
    head->payload = 'A';
    head->next = NULL;
    insert_after(head, 'B');
    insert_after(head->next, 'C');
    // check for errors (omitting for brevity)
    print_list(head);
    // free everything (we'll see how later)
    return 0;
}
```

Linked lists: inserting

```
$ gcc -c ll_insert.c -std=c99 -pedantic -Wall -Wextra  
$ gcc -o ll_insert ll_insert.o  
$ ./ll_insert  
A B C
```

Linked lists: inserting

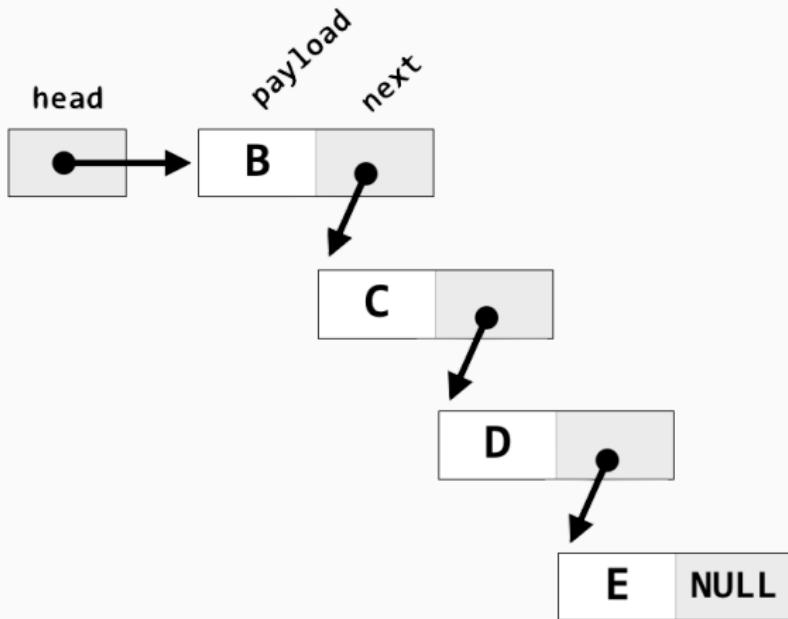
Insertion steps:

- Allocate new node
- Make new node's `->next` point to successor
- Make predecessor's `->next` point to new node

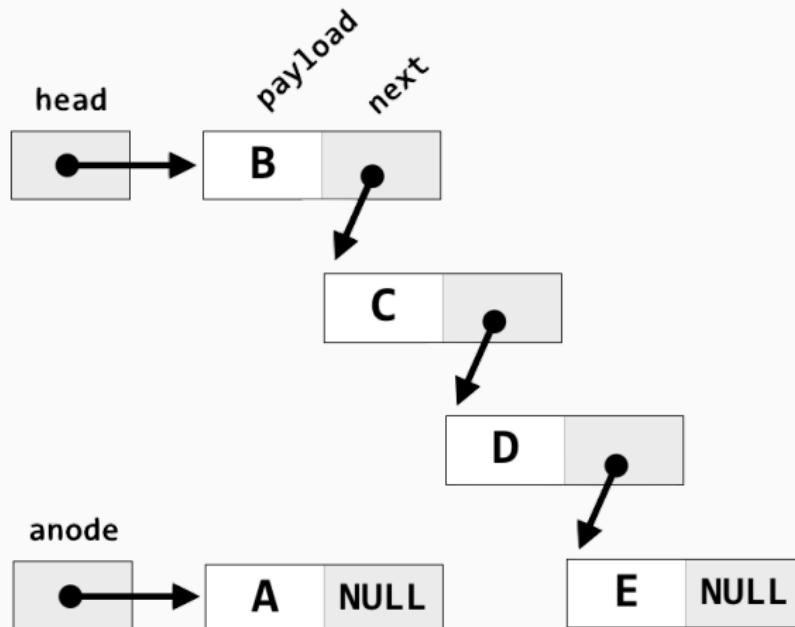
Special cases:

- Inserting at the end (new node's `->next` gets `NULL`)
- Inserting at the beginning. . .

Linked lists: inserting

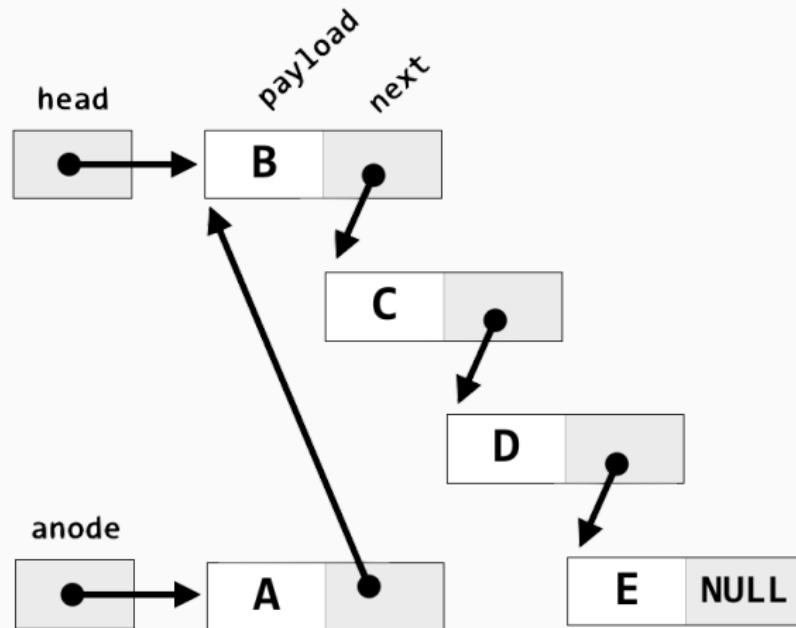


Linked lists: inserting



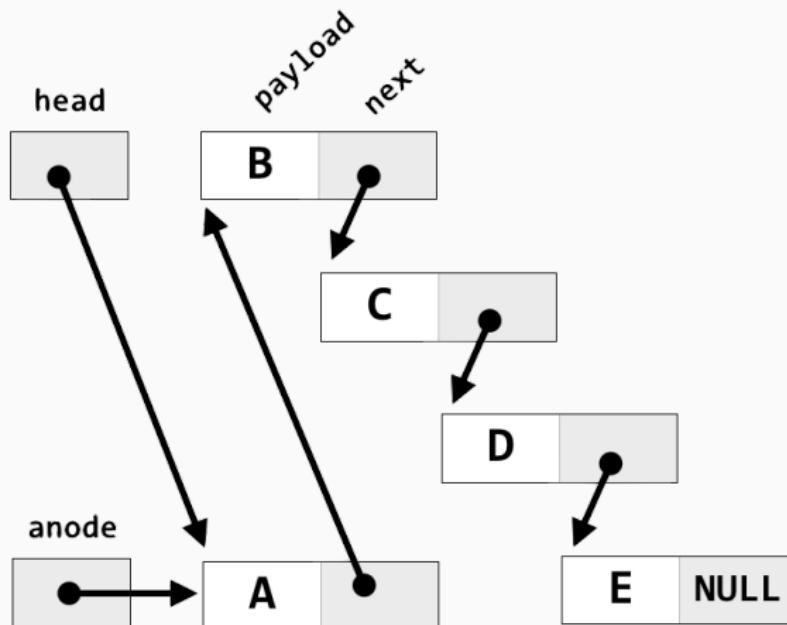
```
struct Node *anode = malloc(sizeof(struct node));
anode->payload = 'A';
anode->next = NULL;
```

Linked lists: inserting



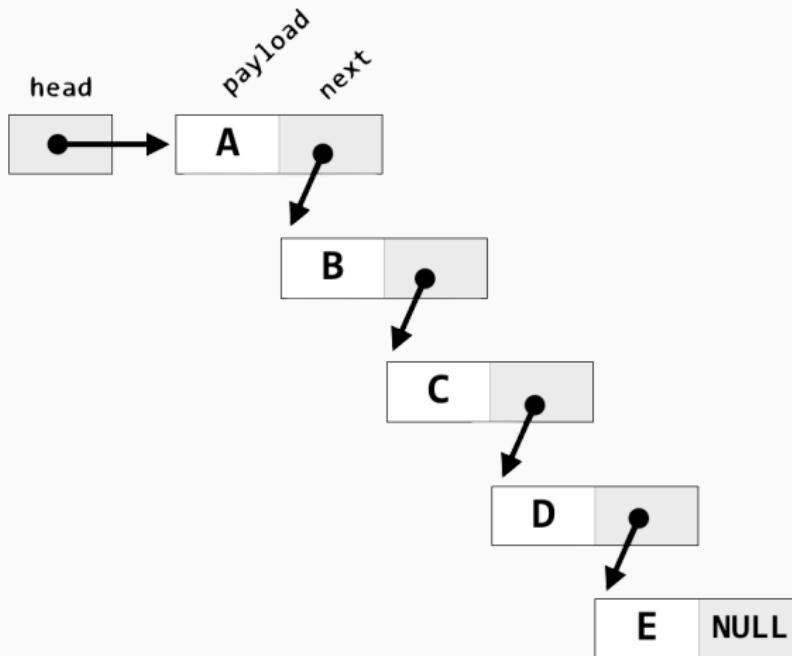
```
anode->next = head;
```

Linked lists: inserting



```
head = anode;
```

Linked lists: inserting



Note this involves changing the head pointer. If the caller or any other function maintains a head pointer, they need to update it!

Linked lists: traversing

We already saw a function that *traverses* the list nodes in order, printing each payload as it goes:

```
void print_list(Node *cur) {  
    while(cur != NULL) {  
        printf("%c ", cur->payload);  
        cur = cur->next;  
    }  
    putchar('\n');  
}
```

Linked lists: traversing

Such functions tend to follow this form:

```
??? do_something(Node *cur) {  
    ???  
    while(cur != NULL) {  
        ???  
        cur = cur->next;  
    }  
    return ???;  
}
```

Linked lists: traversing

Complete this function

```
int list_length(Node *cur) {  
    ???  
    while(cur != NULL) {  
        ???  
        cur = cur->next;  
    }  
    return ???;  
}
```

Linked lists: traversing

Complete this function

```
int list_length(Node *cur) {  
    int length = 0;  
    while(cur != NULL) {  
        length++;  
        cur = cur->next;  
    }  
    return length;  
}
```

Linked lists: traversing

Can we complete this function?

```
void free_list(Node *cur) {  
    while(cur != NULL) {  
        ???  
    }  
}
```

Note: to free a list, it's not enough to free the head. That "leaks" the non-head nodes.

Linked lists: traversing

Careful: we have to avoid freeing a Node *before* we've followed its next pointer

We can't dereference already-freed memory

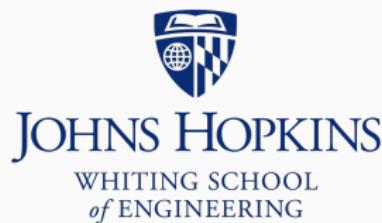
Linked lists: traversing

```
void free_list(Node *cur) {
    while(cur != NULL) {
        Node *next = cur->next; // *first* get `next`
        free(cur);             // *then* free `cur`
        cur = next;
    }
}
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++ Introduction

If learning C is like learning “business English,” learning C++ is like learning the rest of English



C++

Sometimes programming in C is the best or only option

- You “inherited” C code
- No C++ compiler is available for system you’re targeting
- Your software must work closely with the Linux kernel, or other C-based software

If we started a new project today, especially if it was big or involved many people, we’d probably choose C++

C++

Classes – like Java classes

Templates – like Java generics

Standard Template Library – like `java.util`

More convenient text input & output

C++

C++ is not a “superset” of C; many C programs don’t immediately work in C++

Think of C and C++ as closely related but different languages

Most concepts and constructs in C work *the same way* in C++:

- Types: int, char, char *, etc
 - C++ adds bool (equals either true or false)
- Numeric representations & properties
- Arrays, pointers, * and &, pointer arithmetic
- if/else if/else, switch/case, for, while, do/while
- Pass by value (still the default), pass by pointer
- Stack vs. heap, scope & lifetime
- Operators: arithmetic, relational, logical, assignment, bitwise
- struct (minor differences)
- Casting (minor differences)

C++

Our favorite tools work just as well with C++:

- git
- make
- gdb
- valgrind

C++

```
#include <iostream>

using std::cout;
using std::endl;

int main() {
    cout << "Hello world!" << endl;
    return 0;
}

$ g++ -c hello_world.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o hello_world hello_world.o
$ ./hello_world
Hello world!
```

C++

Programming stages same as for C: edit -> compile -> execute

When compiling:

- g++ instead of gcc
- -std=c++11 instead of -std=c99
- .cpp instead of .c

```
$ g++ -c hello_world.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o hello_world hello_world.o  
$ ./hello_world  
Hello world!
```

Options we used with gcc work with g++ too

- -o to set name of executable
- -c to compile to .o file
- -g to include debug symbols
- -Wall -Wextra -pedantic for sensitive warnings & errors

C++: libraries

```
#include <iostream>
```

As with C, C++ library headers are included with < angle brackets >

For standard C++ headers, *omit the trailing .h*

- <iostream>, not <iostream.h>

```
#include "linked_list.h"
```

User-defined headers use " quotes " and end with .h as usual

- You'll sometimes see .hpp instead of .h, but we'll use .h

C++: libraries

Can use familiar C headers: assert.h, math.h, ctype.h, stdlib.h,

...

When #include'ing, drop .h & add c at the beginning:

```
#include <iostream>
#include <cassert> // dropped .h, added c at beginning

using std::cout;
using std::endl;

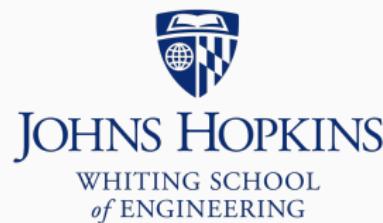
int main(int argc, char *argv[]) {
    assert(argc > 1); // our old friend assert
    cout << "Hello " << argv[1] << "!" << endl;
    return 0;
}
```

C++ I/O and namespaces

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++ I/O and namespaces

We saw this example:

```
#include <iostream>
#include <cassert> // dropped .h, added c at beginning

using std::cout;
using std::endl;

int main(int argc, char *argv[]) {
    assert(argc > 1); // our old friend assert
    cout << "Hello " << argv[1] << "!" << endl;
    return 0;
}
```

C++: I/O

```
$ g++ -c hello_world_2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o hello_world_2 hello_world_2.o  
$ ./hello_world_2 Everyone  
Hello Everyone!
```

Note: argc and argv work just like in C

C++: I/O

iostream is the main C++ library for input and output

```
#include <iostream>
```

C++: namespaces

```
using std::cout;  
using std::endl;
```

C++ has *namespaces*.

- In C, when two things have the same name, we get errors (from compiler or linker) and confusing situations (“shadowing”)
- In C++, items with same name can safely be placed in distinct “namespaces”, similar to Java packages / Python modules

C++: namespaces

Most C++ functionality lives in namespace called std

If we didn't include:

```
using std::cout;  
using std::endl;
```

at the top, then we would have to write the fully qualified name each time:

```
std::cout << "Hello world" << std::endl;
```

Do not use using in a header file

Doing so affects all the source files that include that header, even indirectly, which can lead to confusing name conflicts

- *Only* use using in source .cpp files
- This will be enforced in homework grading

Use fully qualified names (e.g. std::endl) in headers

C++: namespaces

Do not use using namespace <id>

This is bad practice as it brings *all* the names in namespace <id> into the current namespace

You will probably see C++ code with using namespace std; don't imitate this. You will lose points if you do this on homeworks.

C++: I/O

Text input & output in C++ are simpler than in C, thanks to C++'s stream operators and libraries

(We will probably not cover binary I/O in C++)

C++: I/O

```
cout << "Hello world!" << endl;
```

cout is our old friend, the standard output stream

- Like stdout in C

endl is the newline character

- C++ has '\n' too, but endl is usually preferred

<< is the *insert operator*

- Also known as *insertion operator* or *output operator*

C++: I/O

Insert operator joins all the items to write in a “chain”

Leftmost item in chain is the stream being written to

```
cout << "We have " << inventory << " " << item << "s left,"  
    << " costing $" << price << " per unit" << endl;
```

cout is the stream, everything else is the string to write to it

C++: I/O

```
#include <iostream>
using std::cout;
using std::endl;

int main() {
    int inventory = 44;
    double price = 70.07;
    const char *item = "chainsaw";
    bool on_sale = true;

    cout << "We have " << inventory << " " << item << "s left,"
        << " costing $" << price << " per unit."
        << " On sale = " << on_sale << '.' << endl;
    return 0;
}
```

```
$ g++ -c cpp_io_1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o cpp_io_1 cpp_io_1.o
$ ./cpp_io_1
We have 44 chainsaws left, costing $70.07 per unit. On sale = 1.
```

C++: I/O

No format specifiers (%d, %s etc)

Instead, items to be printed are arranged in printing order; easier to read and understand

```
int inventory = 44;
double price = 70.07;
const char *item = "chainsaw";
bool on_sale = true; // *** note the "bool" type

cout << "We have " << inventory << " " << item << "s left,"
    << " costing $" << price << " per unit."
    << " On sale = " << on_sale << '.' << endl;
```

Note that bool is printed like an integer

C++: I/O

An example of C++ I/O but also an example of *operator overloading*

`<<` usually does bitwise left-shift; but if operand on the left is a C++ stream (`cout`), `<<` is the insert operator

```
cout << "Hello world!" << endl;
```

More on this later

C++: I/O

How much of the C library can we use in C++? Most of it.

```
#include <cstdio>

int main() {
    int inventory = 44;
    double price = 70.07;
    const char *item = "chainsaw";
    int on_sale = 1; // *** note the "bool" type

    printf("We have %d %ss left, costing $%f per unit. On sale = %d.\n",
           inventory, item, price, on_sale);
    return 0;
}
```

```
$ g++ -c cpp_io_2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o cpp_io_2 cpp_io_2.o
$ ./cpp_io_2
We have 44 chainsaws left, costing $70.070000 per unit. On sale = 1.
```

C++: I/O

```
#include <iostream>
#include <string> // new header -- not used in C

using std::cout;
using std::cin;
using std::endl;
using std::string;

int main() {
    cout << "Please enter your first name: ";
    string name;
    cin >> name; // read user input into string object
    cout << "Hello, " << name << "!" << endl;
    return 0;
}
```

C++: I/O

```
$ g++ -c cpp_io_3.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o cpp_io_3 cpp_io_3.o  
$ echo Ed | ./cpp_io_3  
Please enter your first name: Hello, Ed!
```

C++: I/O

```
cin >> name;
```

Reads one whitespace-delimited token from standard input and places the result in string name

>> is the *extraction operator*

- Also called *input operator*

C++: I/O

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

using std::cout;
using std::cin;
using std::endl;
using std::string;

int main() {
    string word, smallest;
    while(cin >> word) {
        if(smallest.empty() || word < smallest) {
            smallest = word;
        }
    }
    cout << smallest << endl;
    return 0;
}
```

C++: I/O

```
$ g++ -c smallest_word.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o smallest_word smallest_word.o  
$ echo "the quick brown fox" | ./smallest_word  
brown
```

C++: I/O

```
while(cin >> word) {  
    // ...  
}
```

cin >> word evaluates to true if the input stream is still in a “good state” (no error, no EOF) after reading the word

C++: I/O

```
#include <iostream>
#include <cctype>

using std::cout;
using std::cin;
using std::endl;

int main() {
    char ch;
    while(cin.get(ch)) { // read single character
        ch = toupper(ch);
        cout << ch; // print single character
    }
    cout << endl;
    return 0;
}
```

C++: I/O

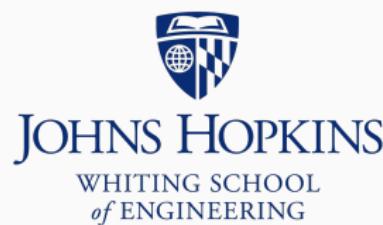
```
$ g++ -c uppercase.cpp.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o uppercase_cpp uppercase_cpp.o  
$ echo "The Quick Brown Fox" | ./uppercase_cpp  
THE QUICK BROWN FOX
```

C++ Strings

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++ Strings

C++ strings have similar user-friendliness of Java/Python strings

Spare us from details like null terminators

(We will still need C strings sometimes, e.g. `char *argv[]`)

C++: string

Use `#include <string>` to use C++ strings

Full name is `std::string`; or put using `std::string`; at the top of .cpp file

C++: string

`s[5]` accesses 6th character in string

`s.at(5)` does the same, additionally doing a “bounds check”

- Like Java’s `ArrayIndexOutOfBoundsException` or Python’s `IndexError`

C++: string

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

using std::cout;
using std::endl;
using std::string;

int main() {
    string s("Nobody's perfect");
    for(size_t pos = 0; pos <= s.length(); pos++) { // too far
        cout << s.at(pos);
    }
    cout << endl;
    return 0;
}
```

C++: string

```
$ g++ -c string_at.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o string_at string_at.o
$ ./string_at
terminate called after throwing an instance of 'std::out_of_range'
what(): basic_string::at: __n (which is 16) >= this->size() (which is 16)
```

Better to use `assert(...)` to check that you're in-bounds. But `s.at(x)` (instead of `s[x]`) is another way to be cautious.

C++: string

Some ways to initialize a new string variable:

```
string s1 = "world";    // initializes to "world"
string s2("hello");    // just like s2 = "hello"
string s3(3, 'a');      // s2 is "aaa"
string s4;              // empty string ""
string s5(s2);          // copies s2 into s5
```

C++: string

strings can be arbitrarily long

The C++ library worries about the memory

- Dynamically allocated and adjusted as needed
- When string goes out of scope, memory is freed

Automatic handling of heap memory is a major advantage of C++

- We will leverage it for our own classes later

C++: string

Assuming s, s1 and s2 are std::string's:

```
s = "wow"           // assign literal to string
cin >> s            // put one whitespace-delimited input word in s
cout << s            // write s to standard out
getline(cin, s)     // read to end of line from stdin, store in s
s1 = s2              // copy contents of s2 into s1
s1 + s2              // return new string: s1 concatenated with s2
s1 += s2             // same as s1 = s1 + s2, also same as s1.append(s2)
== != < > <= >=    // relational operators; alphabetical order
```

C++: string

```
string s = "hello";
cout << s.length() << endl; // prints 5

// prints bytes of memory allocated
cout << s.capacity() << endl;

// s.substr(offset, howmany) gives substring of s
cout << s.substr(1, 3) << endl; // prints "ell"

// s.c_str() returns C-style "const char *" version
cout << strlen(s.c_str()) << endl; // prints 5
```

C++: string

See C++ reference for more string functionality

- www.cplusplus.com/reference/string/string/

Commonly used member functions (click for links):

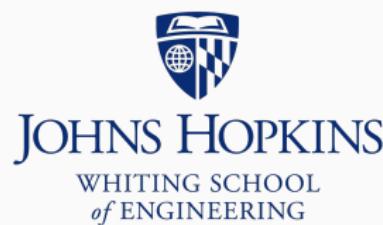
- `length` – return # of characters (ignoring terminator)
- `empty` – return true when there is at least 1 character
- `append` – like `+=`
- `push_back` – like append for a single character
- `clear` – set to empty string
- `insert` – insert one string in middle of another
- `erase` – remove stretch of characters from string
- `replace` – replace a substring with a given string

C++ STL Introduction

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++ STL Introduction

Standard Template Library (STL) is C++'s library of useful data structures & algorithms

- Like `java.util/java.lang`
- Like Python sets, dictionaries, collections

Templates are covered in detail later in the course; we'll give them a quick look now

C++: Templates

Templates are a way of writing an object (Node) or function (print_list) to work with *any* type

You simultaneously define a *family* of related objects/functions

C++: Templates

```
struct Node {  
    T payload; // 'T' is placeholder for a type  
    Node *next;  
};  
  
void print_list(Node *head) {  
    Node *cur = head;  
    while(cur != NULL) {  
        cout << cur->payload << " ";  
        cur = cur->next;  
    }  
    cout << endl;  
}
```

We could replace T with int, float, char, or std::string and this would compile & work

C++: Templates

```
template<typename T>
struct Node {
    T payload;
    Node *next;
};

template<typename T>
void print_list(Node<T> *head) {
    Node<T> *cur = head;
    while(cur != NULL) {
        cout << cur->payload << " ";
        cur = cur->next;
    }
    cout << endl;
}
```

Same example, using templates

One struct/function, works for (almost) *any* type T

C++: Template primer

```
int main() {
    Node<float> f3 = {95.1f, NULL}; // float payload
    Node<float> f2 = {48.7f, &f3}; // float payload
    Node<float> f1 = {24.3f, &f2}; // float payload
    print_list(&f1);

    Node<int> i2 = {239, NULL}; // int payload
    Node<int> i1 = {114, &i2}; // int payload
    print_list(&i1);

    return 0;
}

$ g++ -c ll_template_cpp.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ll_template_cpp ll_template_cpp.o
$ ./ll_template_cpp
24.3 48.7 95.1
114 239
```

C++: STL

With STL we use types like `vector<string>`

We read that type as “*a vector of strings*”

- `vector<string>` – a vector of `std::strings`
- `vector<float>` – a vector of floats
- `map<string, int>` – a structure that maps `std::strings` to ints

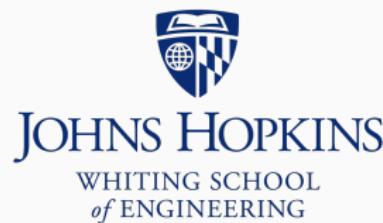
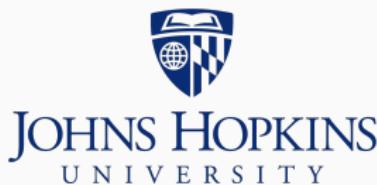
Similar to Java generics

C++: vector & iterators

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++: vector & iterators

vector is an array that automatically grows/shrinks as you need more/less room

- Use [x] or .at(x) to access an element, like std::string
 - .at(x) does bounds check, like std::string
- Allocation, resizing, deallocation handled by C++
- Like Java's java.util.ArrayList or Python's list type

#include <vector> to use it

std::string is like (but not same as) std::vector<char>

C++: vector

Declare a vector:

```
using std::vector;  
vector<std::string> names;
```

Add elements to vector (at the back):

```
names.push_back("Alex Hamilton");  
names.push_back("Ben Franklin");  
names.push_back("George Washington");
```

Print number of items in vector, and first and last items:

```
cout << "Size=" << names.size()  
<< ", first=" << names.front()  
<< ", last=" << names.back() << endl;
```

C++: vector

vector handles memory for you

Behind the scenes, dynamic memory allocations are needed both to create strings and to add them to the growing vector:

```
names.push_back("Alex Hamilton");
names.push_back("Ben Franklin");
names.push_back("George Washington");
```

Allocations happen automatically; everything (vector & strings) is deallocated when names goes out of scope

C++: vector

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

using std::vector; using std::string;
using std::cin;    using std::cout;
using std::endl;

int main() {
    vector<string> names;
    names.push_back("Alex Hamilton");
    names.push_back("Ben Franklin");
    names.push_back("George Washington");

    cout << "First name was " << names.front() << endl;
    cout << "Last name was " << names.back() << endl;
    // names.front() is like names[0]
    // names.back() is like names[names.size()-1]

    return 0;
} // names goes out of scope and memory is freed
```

C++: vector

```
$ g++ -c names_1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o names_1 names_1.o  
$ ./names_1  
First name was Alex Hamilton  
Last name was George Washington
```

C++: vector

Two ways to print all elements of a vector. With indexing:

```
for(size_t i = 0; i < names.size(); i++) {  
    cout << names[i] << endl;  
}
```

With an *iterator*:

```
for(vector<string>::iterator it = names.begin();  
    it != names.end();  
    ++it)  
{  
    cout << *it << endl;  
}
```

C++: vector

Iterators are “clever pointers” that know how to move over the components of a data structure

Structure could be simple (linked list) or complicated (tree)

They are safer & less error-prone than pointers; pointers cannot generally be used with STL containers

C++: iterators

For STL container of type T, iterator has type T::iterator

```
for(vector<string>::iterator it = names.begin();
    it != names.end();
    ++it)
{
    cout << *it << endl;
}
cout << endl;
```

Here, iterator type is vector<string>::iterator

C++: iterators

Looking harder at the loop:

```
for(vector<string>::iterator it = names.begin();  
    it != names.end();  
    ++it)
```

First line: declares it, sets it to point to first element initially

Second: stops loop when iterator has moved past vector end

Third: tells iterator to advance by 1 each iteration

- `++it` isn't really pointer arithmetic; `++` is "overloaded" to move forward 1 element *like* a pointer

C++: iterators

Looking harder at the body:

```
cout << *it << endl;
```

*it is *like* dereferencing; * is “overloaded” to get the element currently pointed to by the iterator

For vector, *it's type equals the element type, string in this case

C++: vector

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

using std::vector; using std::string;
using std::cin;    using std::cout;
using std::endl;

int main() {
    vector<string> names;
    names.push_back("Alex Hamilton");
    names.push_back("Ben Franklin");
    names.push_back("George Washington");
    for(vector<string>::iterator it = names.begin();
        it != names.end();
        ++it)
    {
        cout << *it << endl;
    }
    return 0;
}
```

C++: vector

```
$ g++ -c names_2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o names_2 names_2.o  
$ ./names_2  
Alex Hamilton  
Ben Franklin  
George Washington
```

C++: vector

Iterate in *reverse* order by using `T::reverse_iterator`, `.rbegin()` and `.rend()` instead:

```
for(vector<string>::reverse_iterator it = names.rbegin();
    it != names.rend();
    ++it)
{
    cout << *it << endl;
}
```

```
$ g++ -c names_3.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o names_3 names_3.o
$ ./names_3
George Washington
Ben Franklin
Alex Hamilton
```

C++: vector

See C++ reference for more vector functionality

- www.cplusplus.com/reference/vector/vector/

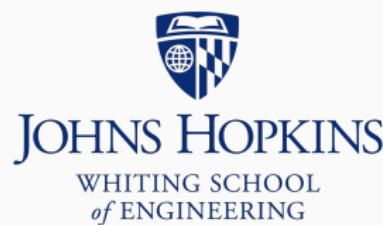
Don't miss:

- `front` – get first element
- `back` – get last element
- `pop_back` – return and delete final element
- `begin/end` – iterators for beginning/end
- `cbegin/cend` – const_iterators for beginning/end
- `rbegin/rend` – reverse_iterators for beginning/end
- `erase, insert, clear, at, empty` – just like string

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++: map

Collection of *keys*, each with an associated *value*

- Like Java's `java.util.HashMap` or `TreeMap`
- Like Python's `dict` (dictionary) type

Value can be any type you wish (even another container)

Key can be any type where `<` can be used to compare values

- All numeric types, `char`, `std::string`, etc

C++: map

Declare a map:

```
map<int, string> jhed_to_name;
```

Add a key + value to a map:

```
jhed_to_name[92394] = "Alex Hamilton";
```

Print a key and associated value:

```
const int k = 92394;  
cout << "Key=" << k << ", Value=" << jhed_to_name[k] << endl;
```

C++: map

A map can only associate 1 value with a key

```
const int k = 92394;  
jhed_to_name[k] = "Alex Hamilton";  
jhed_to_name[k] = "George Washington"; // Alex is replaced  
cout << k << ":" << jhed_to_name[k] << endl;
```

```
$ g++ -c jhed_map_0.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o jhed_map_0 jhed_map_0.o  
$ ./jhed_map_0  
92394: George Washington
```

C++: map

Get number of keys:

```
jhed_to_name.size();
```

Check if map contains (has a value for) a given key:

```
if(jhed_to_name.find(92394) != jhed_to_name.end()) {
    cout << "Found it" << endl;
} else {
    cout << "Didn't find it" << endl;
}
```

C++: vector

To get all the elements of the map, use an *iterator*:

```
for(map<int, string>::iterator it = jhed_to_name.begin();
    it != jhed_to_name.end();
    ++it)
{
    cout << " " << it->first << ":" << it->second << endl;
}
```

Iterator type: `map<int, string>::iterator`

Loop is similar to the loop for vector

Keys iterated over *in ascending order* (increasing id in this case)

C++: vector

Looking at the body:

```
cout << " " << it->first << ":" " << it->second << endl;
```

Dereferenced map iterator type is std::pair<key_type,
value_type>

- `it->first` is the key (int here)
- `it->second` is the value (string here)

More on pair later

C++: map

```
#include <iostream>
#include <map>
#include <string>
using std::cout;   using std::endl;
using std::string; using std::map;

int main() {
    map<int, string> jhed_to_name;
    jhed_to_name[92394] = "Alex Hamilton";
    jhed_to_name[13522] = "Ben Franklin";
    jhed_to_name[42345] = "George Washington";
    cout << "size of jhed_to_name " << jhed_to_name.size() << endl;
    cout << "jhed_to_name[92394] " << jhed_to_name[92394] << endl;
    for(map<int, string>::iterator it = jhed_to_name.begin();
        it != jhed_to_name.end();
        ++it)
    {
        cout << " " << it->first << ":" << it->second << endl;
    }
    return 0;
}
```

C++: map

```
$ g++ -c jhed_map.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o jhed_map jhed_map.o  
$ ./jhed_map  
size of jhed_to_name 3  
jhed_to_name[92394] Alex Hamilton  
13522: Ben Franklin  
42345: George Washington  
92394: Alex Hamilton
```

Note again: the keys are printed *in ascending order*

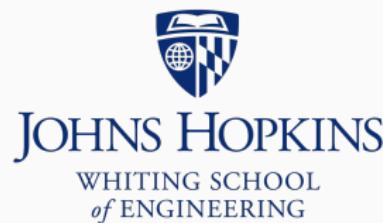
Can use `reverse_iterator`, `.rbegin()` and `.rend()` to get keys in *descending order*

C++: More on iterators

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++: More on iterators

Consider these map iterator types:

- `map<int, string>::iterator` – iterator over a map
- `map<string, map<string, int> >::iterator` – iterator over a map *where the values are themselves maps*

`typedef` can help by:

- Reducing clutter
- Bringing related type declarations closer together in your code:

```
typedef map<int, string> TMap;      // map type
typedef TMap::iterator TMapItr;       // map iterator type
```

C++: Iterators

With iterator (or reverse_iterator) you can modify the data structure via the dereferenced iterator:

```
typedef vector<int>::iterator TIter;

void prefix_sum(TIter begin, TIter end) {
    int sum = 0;
    for(TIter it = begin; it != end; ++it) {
        *it += sum;
        sum += *it;
    }
}
```

```
$ g++ -c prefix_sum_iter.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o prefix_sum_iter prefix_sum_iter.o
$ ./prefix_sum_iter
Before: 1 1 1 1
After: 1 2 4 8
```

C++: Iterators

`const_iterator` does not allow modifications

```
typedef vector<int>::const_iterator TIter;
//           ^^^^^^

void prefix_sum(TIter begin, TIter end) {
    int sum = 0;
    for(TIter it = begin; it != end; ++it) {
        *it += sum;
        sum += *it;
    }
}
```

```
$ g++ -c prefix_sum_iter.cpp -std=c++11 -pedantic -Wall -Wextra
prefix_sum_iter.cpp: In function 'void prefix_sum(TIter, TIter)': 
prefix_sum_iter.cpp:13:16: error: assignment of read-only location
'it._ZSt13normal_iteratorISt11vectorIiESt11vectorIiEoperator*()'
    *it += sum;
           ^~~~
```

C++: Iterators

Type	++it	--it	Get with	*it type
iterator	forward	back	.begin()/end()	-
const_iterator	forward	back	.cbegin()/cend()	const
reverse_iterator	back	forward	.rbegin()/rend()	-
const_reverse_iterator	back	forward	.crbegin()/crend()	const

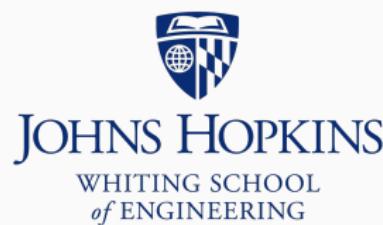
Reminder: When using a variant of `.begin()/.end()`, always initialize the iterator with `.begin()` and stop as soon as it == `.end()`. Don't switch the two, even if you're *trying* to go backwards. Use the `reverse_` version instead.

C++: pair and tuple

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++: pair and tuple

We want a function that takes integers a & b and returns both a/b (quotient) and $a\%b$ (remainder)

- `divmod(10, 5)` returns $2, 0$
- `divmod(10, 3)` returns $3, 1$

Functions only return *one* thing

C++: pair

One solution: pass-by-pointer arguments:

```
void divmod(int a, int b, int *quo, int *rem) {  
    *quo = a / b;  
    *rem = a % b;  
}
```

Another: define struct for divmod's return type:

```
struct quo_rem {  
    int quotient;  
    int remainder;  
};  
  
quo_rem divmod(int a, int b) { ... }
```

C++: pair

STL solution: return pair<int, int>, a pair with items of types int & int:

```
pair<int, int> divmod(int a, int b) {
    return make_pair(a/b, a%b);
}

int main() {
    pair<int, int> qr_10_5 = divmod(10, 5);
    pair<int, int> qr_10_3 = divmod(10, 3);
    cout << "10/5 quotient=" << qr_10_5.first
        << ", remainder=" << qr_10_5.second << endl;
    cout << "10/3 quotient=" << qr_10_3.first
        << ", remainder=" << qr_10_3.second << endl;
    return 0;
}
```

C++: pair

We've used pair already; dereferenced map iterator is a pair:

```
for(map<int, string>::iterator it = jhed_to_name.begin();
    it != jhed_to_name.end();
    ++it)
{
    // it->first has type int
    // it->second has type string
    cout << " " << it->first << ":" << it->second << endl;
}
```

C++: pair

Relational operators for pair work as expected:

- Compares first field first...
- ...if there's a tie, compares second field

`make_pair(2, 3) < make_pair(3, 2)` is true

C++: tuple

tuple is like pair but with as many fields as you like

```
#include <tuple>
using std::tuple; using std::make_tuple;

tuple<int, int, float> divmod(int a, int b) {
    return make_tuple(a/b, a%b, (float)a/b);
}
```

std::get<N>(tup) gets the Nth field of tuple called tup:

```
using std::get;
tuple<int, int, float> tup = divmod(10, 3);
cout << "10/3 quotient="      << get<0>(tup)
    << ", remainder="        << get<1>(tup)
    << ", decimal quotient=" << get<2>(tup) << endl;
```

C++: tuple

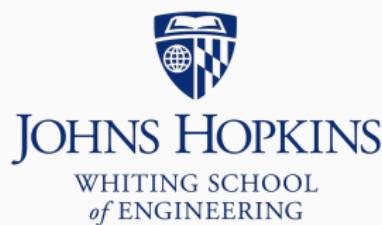
```
$ g++ -c quo_rem_tuple.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o quo_rem_tuple quo_rem_tuple.o  
$ ./quo_rem_tuple  
10/3 quotient=3, remainder=1, decimal quotient=3.33333
```

References

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

References

Like iterators, references provide pointer-like functionality without many of the pointer drawbacks

Reference variable is an *alias*, another name for an existing variable

Restrictions make them safer than pointers:

- Can't be NULL
- Must be initialized immediately
- Once set to alias a variable, cannot later alias a different variable

Because of these, it's harder (though still possible) to access memory that doesn't belong to you via reference

C++: References

A *reference* to variable of type int has type int&

- & is *part of the type*
- Reminds us of “address of” operator &

C++: References

Example with pointer:

```
#include <iostream>
using std::cout;  using std::endl;

int main() {
    int i = 1;
    int *j = &i;
    cout << "i=" << i << ", *j=" << *j << endl;

    *j = 9;
    cout << "i=" << i << ", *j=" << *j << endl;
    return 0;
}

$ g++ -c ref_ptr_1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref_ptr_1 ref_ptr_1.o
$ ./ref_ptr_1
i=1, *j=1
i=9, *j=9
```

C++: References

Same example with reference:

```
#include <iostream>
using std::cout;  using std::endl;

int main() {
    int i = 1;
    int& j = i; // j is an alias (another name) for i
    cout << "i=" << i << ", j=" << j << endl;

    j = 9; // no dereference (*) needed
    cout << "i=" << i << ", j=" << j << endl;
    return 0;
}

$ g++ -c ref_ptr_2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref_ptr_2 ref_ptr_2.o
$ ./ref_ptr_2
i=1, j=1
i=9, j=9
```

C++: References

You can get the address of a reference variable with &; it has the same address as the variable it aliases

```
#include <iostream>
using std::cout;  using std::endl;

int main() {
    int a = 5;
    int& b = a;
    cout << "&a=" << &a << endl << "&b=" << &b << endl;
    return 0;
}
```

```
$ g++ -c ref_addr.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref_addr ref_addr.o
$ ./ref_addr
&a=0x7ffeb7dacc94
&b=0x7ffeb7dacc94
```

C++: References

Function parameters with reference type are passed “by reference”

```
// if you have int a = 1, b = 2;  
// then call like this: swap(a, b) -- no &s!  
void swap(int& a, int& b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}
```

Passing by reference is like passing by pointer

- Callee can modify variable in caller
- Large data structures can be passed without copying

C++: References

```
#include <iostream>
using std::cout;  using std::endl;

void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main() {
    int a = 1, b = 9;
    swap(a, b);
    cout << "a=" << a << ", b=" << b << endl;
    return 0;
}

$ g++ -c ref_swap.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref_swap ref_swap.o
$ ./ref_swap
a=9, b=1
```

C++: References

Recall this example; ch passed by reference to cin.get(char&)

```
#include <iostream>
#include <cctype>
using std::cout;  using std::endl;

int main() {
    char ch;
    // read standard input char by char
    while(cin.get(ch)) { // pass ch by reference!
        cout << toupper(ch);
    }
    cout << endl;
    return 0;
}
```

C++: References

C++ has *both* pass-by-value (non-reference parameters) *and* pass-by-reference (reference parameters)

Function can have a mix of pass-by-value and pass-by-reference parameters

C++: References

```
#include <iostream>
using std::cout;  using std::endl;

// `int a` and `int b` are passed *by value*
// `int& quo` and `int& rem` are passed *by reference*
void divmod(int a, int b, int& quo, int& rem) {
    quo = a / b;
    rem = a % b;
}

int main() {
    int a = 10, b = 3, quo, rem;
    divmod(a, b, quo, rem);
    cout << "a=" << a << ", b=" << b
        << ", quo=" << quo << ", rem=" << rem << endl;
    return 0;
}
```

C++: References

```
$ g++ -c divmod_ref.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o divmod_ref divmod_ref.o  
$ ./divmod_ref  
a=10, b=3, quo=3, rem=1
```

C++: References

Looking at the function call doesn't tell you which parameters are passed by value or by reference:

```
divmod(a, b, quo, rem); // ???
```

You have to look at the callee's parameter types:

```
void divmod(int a, int b, int& quo, int& rem) {  
    ...  
}
```

C++: References

We can return a reference

```
#include <iostream>
using std::cout;  using std::endl;

int& minref(int& a, int& b) {
    if(a < b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int a = 5, b = 10;
    int& min = minref(a, b);
    min = 12;
    cout << "a=" << a << ", b=" << b << ", min=" << min << endl;
    return 0;
}
```

C++: References

```
$ g++ -c ref_min.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o ref_min ref_min.o  
$ ./ref_min  
a=12, b=10, min=12
```

minref returns a reference to a

min = 12 modifies both min and a

What if we make minref's arguments non-references?

C++: References

```
#include <iostream>
using std::cout;  using std::endl;

int& minref(int a, int b) {
    if(a < b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int a = 5, b = 10;
    int& min = minref(a, b);
    min = 6;
    cout << "a=" << a << ", b=" << b << ", min=" << min << endl;
    return 0;
}
```

C++: References

```
$ g++ -c %PREV% -std=c++11 -pedantic -Wall -Wextra  
ref_min_2.cpp: In function 'int& minref(int, int)':  
ref_min_2.cpp:4:17: warning: reference to local variable 'a' returned  
    int& minref(int a, int b) {  
        ^  
ref_min_2.cpp:4:24: warning: reference to local variable 'b' returned  
    int& minref(int a, int b) {  
        ^
```

C++: References

Returning a reference to a local variable is just as bad as returning a pointer to one. In our original minref function, this was OK because the parameters themselves were references.

```
int& minref(int& a, int& b) {  
    if(a < b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

C++: References

Once a reference is set to alias a variable, it cannot later be set to alias another variable

Example...

C++: References

```
#include <iostream>
using std::cout;  using std::endl;

int main() {
    int a = 5, b = 10;
    int& c = a;
    cout << "a=" << a << ", c=" << c << endl;
    c = b; // this *doesn't* change c to alias b
            // instead, sets c (and a) to 10
    cout << "a=" << a << ", c=" << c << endl;
    return 0;
}
```

```
$ g++ -c ref_reset.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ref_reset ref_reset.o
$ ./ref_reset
a=5, c=5
a=10, c=10
```

C++: References

A reference variable must be initialized immediately

```
#include <iostream>
using std::cout;  using std::endl;

int main() {
    int& a; // won't compile
    int b = 10;
    a = b;
    cout << a << endl;
    return 0;
}
```

```
$ g++ -c ref_delay.cpp -std=c++11 -pedantic -Wall -Wextra
ref_delay.cpp: In function 'int main()':
ref_delay.cpp:5:10: error: 'a' declared as reference but not initialized
    int& a; // won't compile
           ^
```

C++: References

A reference cannot be NULL

```
#include <iostream>
using std::cout;  using std::endl;

int main() {
    int& a = NULL;
    if(a == NULL) {
        cout << "a is NULL" << endl;
    }
    return 0;
}
```

C++: References

```
$ g++ -c ref_null.cpp -std=c++11 -pedantic -Wall -Wextra
ref_null.cpp: In function 'int main()':
ref_null.cpp:5:14: warning: converting to non-pointer type 'int' from NULL
[-Wconversion-null]
    int& a = NULL;
               ^
In file included from /usr/include/_G_config.h:15:0,
                 from /usr/include/libio.h:31,
                 from /usr/include/stdio.h:41,
                 from /usr/include/c++/7/cstdio:42,
                 from /usr/include/c++/7/ext/string_conversions.h:43,
                 from /usr/include/c++/7/bits/basic_string.h:6347,
                 from /usr/include/c++/7/string:52,
                 from /usr/include/c++/7/bits/locale_classes.h:40,
                 from /usr/include/c++/7/bits/ios_base.h:41,
                 from /usr/include/c++/7/ios:42,
                 from /usr/include/c++/7/ostream:38,
                 from /usr/include/c++/7/iostream:39,
                 from ref_null.cpp:1:
ref_null.cpp:5:14: error: cannot bind non-const lvalue reference of type 'int&'
to an rvalue of type 'int'
    int& a = NULL;
               ^
ref_null.cpp:6:13: warning: NULL used in arithmetic [-Wpointer-arith]
  if(a == NULL) {
               ^
               ^~~~~~
```

C++: References

A reference can be const

- const reference cannot be modified after initialization

You may still be able to modify the original variable

- ... if it's non-const, or
- ... via a non-const reference to the same variable

C++: References

```
#include <iostream>
using std::cout;  using std::endl;

int main() {
    int a = 1;
    int& b = a;
    const int& c = a;
    a = 2;
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    b = 3; // this is fine
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    c = 4; // *** not allowed ***
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
    return 0;
}
```

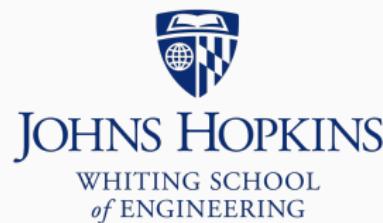
C++: References

```
$ g++ -c ref_const.cpp -std=c++11 -pedantic -Wall -Wextra
ref_const.cpp: In function 'int main()':
ref_const.cpp:12:9: error: assignment of read-only reference 'c'
    c = 4; // *** not allowed ***
           ^
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Constructors

What are the fields of a class set to if we don't explicitly initialize them?

C++ classes

```
#include <vector>

class GradeList {
public:
    void add(double grade) {
        grades.push_back(grade);
        is_sorted = false;
    }

    double percentile(double percentile);
    double mean();
    double median();

private:
    std::vector<double> grades;
    bool is_sorted;
};

};
```

C++ classes

```
int main() {  
    GradeList gl;
```

What are the values of `gl.grades` & `gl.is_sorted` right now?

I haven't set them to anything; are they set to reasonable defaults?

Not in general

- `gl.grades` is initialized properly via its *default constructor* (discussed soon)
- According to C++11 standard, `gl.is_sorted` is *uninitialized!*

Constructors

When we define a class, we can define one or more *constructors*

Each constructor is a way to build a valid instance of that class,
with fields sensibly initialized

If you define no constructors, an *implicit default constructor* is
automatically added by the compiler

Constructors

```
int main() {  
    // behind the scenes, GradeList's implicit  
    // default constructor is called...  
    GradeList gl;  
  
    // ...but it does not initialize gl.is_sorted!  
    ...  
}
```

We will prefer to define the constructor ourselves

Constructors

Terminology:

- *Default constructor* is a constructor that takes no arguments
 - We will see non-default constructors later
- *Implicit default constructor* is the default constructor that the compiler adds when no constructors are specified

Constructors

A constructor is a public member function with the same name as the class

```
class GradeList {  
public:  
    // default constructor for GradeList  
    GradeList() { ... }  
    ...  
}
```

It has no return type; it doesn't return anything

Constructors

We've used default constructors already:

```
// vector<int>'s default constructor initializes empty vector  
vector<int> my_ints;
```

```
// string's default constructor initializes empty string  
string word;
```

Constructors

We cannot call a constructor directly. Constructor is called automatically when a new object is declared, or created using new

```
int main() {  
    // calls default constructor for gl  
    GradeList gl;  
  
    // calls default constructor for *glp  
    GradeList *glp = new GradeList();  
}  
  
(new discussed later)
```

Constructors

Constructors often use a special syntax called an *initializer list*:

```
class GradeList {  
public:  
    // Define our own "default constructor,"  
    GradeList() : grades(), is_sorted(false) { }  
  
    ...  
  
private:  
    std::vector<double> grades;  
    bool is_sorted;  
};
```

Constructors

```
class GradeList {  
public:  
    // Define our own "default constructor,"  
    GradeList() : grades(), is_sorted(false) { }  
    //           ^^^^^^   ^^^^^^  
    //           initializes grades by calling  
    //           std::vector<int>'s default constructor  
    //  
    //           initializes is_sorted by setting it to  
    //           false  
  
    ...  
  
private:  
    std::vector<double> grades;  
    bool is_sorted;  
};
```

Constructors

These default constructors have the same effect:

```
class IntAndString1 {
public:
    IntAndString1() : i(7), s("hello") { }
    //           ^^^^^^^^^^^^^^^^^^^^^^^
    //           "initializer list"
    int i;
    std::string s;
};

class IntAndString2 {
public:
    IntAndString2() {
        i = 7;
        s = "hello";
    }
    int i;
    std::string s;
};
```

Constructors

```
#include <iostream>
#include "def_ctor_1.h"

using std::cout;  using std::endl;

int main() {
    IntAndString1 is1;
    IntAndString2 is2;
    cout << "is1.i=" << is1.i << ", is1.s=" << is1.s << endl;
    cout << "is2.i=" << is2.i << ", is2.s=" << is2.s << endl;
    return 0;
}
```

```
$ g++ -c def_ctor_1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o def_ctor_1 def_ctor_1.o
$ ./def_ctor_1
is1.i=7, is1.s=hello
is2.i=7, is2.s=hello
```

Constructors

Initializer list is the better choice and we will prefer it

- Works as expected both for normal and for reference variables
- Works both for using default and non-default constructors when initializing fields

```
IntAndString() : i(7), s("hello") { }
```

```
IntAndString() {  
    i = 7;  
    s = "hello";  
}
```

Neither Java nor Python have initializer list syntax:

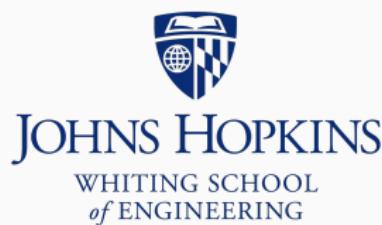
- stackoverflow.com/questions/7154654 in this course; it is clearer & less error-prone to initialize only in constructors

C++ dynamic memory allocation

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

C++ dynamic memory allocation

new and delete are C++ versions of malloc and free

Big difference: new allocates memory *and calls the appropriate constructor*

Small difference: new and delete are “keywords” rather than functions, so we don’t use (...) when calling them

C++ dynamic memory allocation

```
#include <iostream>

using std::cout;  using std::endl;

class DefaultSeven {
public:
    DefaultSeven() : i(7) { }
    int get_i() { return i; }
private:
    int i;
};

int main() {
    DefaultSeven s;
    DefaultSeven *sptr = new DefaultSeven(); // using new
    cout << "s.get_i() = " << s.get_i() << endl;
    cout << "sptr->get_i() = " << sptr->get_i() << endl;
    return 0;
}
```

C++ dynamic memory allocation

```
$ g++ -c new_eg1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o new_eg1 new_eg1.o  
$ ./new_eg1  
s.get_i() = 7  
sptr->get_i() = 7
```

new called the default constructor for us in both cases

C++ dynamic memory allocation

delete deletes something allocated with new

```
int main() {
    DefaultSeven *sptr = new DefaultSeven();
    ... // use sptr
    delete sptr;
    // note: new and delete don't use parentheses,
    // unlike malloc() / free()
    return 0;
}
```

C++ dynamic memory allocation

`T * fresh = new T[n]` allocates an array of n elements of type `T`

Use `delete[]` to deallocate – always use `delete[]` (not `delete`) to deallocate a pointer returned by `new T[n]`

If `T` is a class, then `T`'s default constructor is called for each element allocated

If `T` is a “built-in” type (`int`, `float`, `char`, etc), then the values are *not initialized*, like with `malloc`

C++ dynamic memory allocation

```
#include <iostream>
using std::cout;  using std::endl;

class DefaultSeven {
public:
    DefaultSeven() : i(7) { }
    int get_i() { return i; }
private:
    int i;
};

int main() {
    DefaultSeven *s_array = new DefaultSeven[10];
    for(int i = 0; i < 10; i++) {
        cout << s_array[i].get_i() << " ";
    }
    cout << endl;
    delete[] s_array;
    return 0;
}
```

C++ dynamic memory allocation

```
$ g++ -c new_eg2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o new_eg2 new_eg2.o  
$ ./new_eg2  
7 7 7 7 7 7 7 7 7 7
```

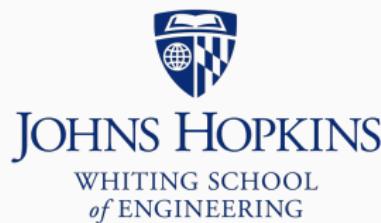
Default constructor was indeed called for all 10 elements

Non-default constructors

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Non-default constructors

Constructors can take parameters, giving the caller more control over how to initialize the object

```
// Uses non-default constructor to initialize s1 to a copy
// of the argument
string s1("hello");

// Same here
string s2 = "world";
```

Non-default constructors

```
#include <iostream>
using std::cout;  using std::endl;

class DefaultSeven {
public:
    DefaultSeven() : i(7) {
        cout << "In default constructor" << endl;
    }

    DefaultSeven(int initial) : i(initial) {
        cout << "In non-default constructor" << endl;
    }

    int get_i() { return i; }

private:
    int i;
};

int main() {
    DefaultSeven s = DefaultSeven(10); // definitely calls non-default ctor
    DefaultSeven s2 = 20; // does this call non-default ctor?
    DefaultSeven *sptr = new DefaultSeven(30); // non-default ctor
    cout << "s.get_i() = " << s.get_i() << endl;
    cout << "s2.get_i() = " << s2.get_i() << endl;
    cout << "sptr->get_i() = " << sptr->get_i() << endl;
    delete sptr;
    return 0;
}
```

Non-default constructors

```
$ g++ -c new_eg3.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o new_eg3 new_eg3.o
$ ./new_eg3
In non-default constructor
In non-default constructor
In non-default constructor
s.get_i() = 10
s2.get_i() = 20
sptr->get_i() = 30
```

Even DefaultSeven s2 = 20; declaration calls non-default constructor

Non-default constructors

If the only constructors provided are non-default, *no implicit default constructor is added* by the compiler

Attempt to use default constructor will yield compiler error

Non-default constructors

```
#include <iostream>
using std::cout;  using std::endl;

class NoDefault {
public:
    NoDefault(int initial) : i(initial) { }

    int get_i() { return i; }

private:
    int i;
};

int main() {
    NoDefault s;
    cout << "s.get_i() = " << s.get_i() << endl;
    return 0;
}
```

Non-default constructors

```
$ g++ -c no_default.cpp -std=c++11 -pedantic -Wall -Wextra
no_default.cpp: In function 'int main()':
no_default.cpp:14:15: error: no matching function for call to
'NoDefault::NoDefault()'
    NoDefault s;
               ^
no_default.cpp:6:5: note: candidate: NoDefault::NoDefault(int)
    NoDefault(int initial) : i(initial) { }
               ^
no_default.cpp:6:5: note:    candidate expects 1 argument, 0 provided
no_default.cpp:4:7: note: candidate: constexpr NoDefault::NoDefault(const
NoDefault&)
class NoDefault {
               ^
no_default.cpp:4:7: note:    candidate expects 1 argument, 0 provided
no_default.cpp:4:7: note: candidate: constexpr NoDefault::NoDefault(NoDefault&&)
no_default.cpp:4:7: note:    candidate expects 1 argument, 0 provided
```

Non-default constructors

```
int main() {
    NoDefault *s = new NoDefault[10];
    cout << "s[9].get_i() = " << s[9].get_i() << endl;
    delete[] s;
    return 0;
}
```

This won't work either; new tries to call default constructor on each allocated NoDefault

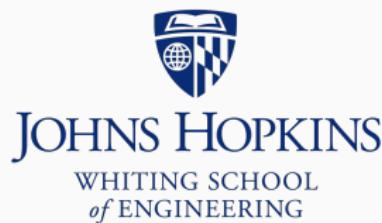
Non-default constructors

```
$ g++ -c no_default2.cpp -std=c++11 -pedantic -Wall -Wextra
no_default2.cpp: In function 'int main()':
no_default2.cpp:14:36: error: no matching function for call to
'NoDefault::NoDefault()'
    NoDefault *s = new NoDefault[10];
                           ^
no_default2.cpp:6:5: note: candidate: NoDefault::NoDefault(int)
    NoDefault(int initial) : i(initial) { }
    ^~~~~~
no_default2.cpp:6:5: note:    candidate expects 1 argument, 0 provided
no_default2.cpp:4:7: note: candidate: constexpr NoDefault::NoDefault(const
NoDefault&)
class NoDefault {
    ^~~~~~
no_default2.cpp:4:7: note:    candidate expects 1 argument, 0 provided
no_default2.cpp:4:7: note: candidate: constexpr
NoDefault::NoDefault(NoDefault&&)
no_default2.cpp:4:7: note:    candidate expects 1 argument, 0 provided
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Destructors

```
#include <cassert>

// What does this class do? Anything wrong with it?
class Sequence {
public:
    Sequence(int sz) : array(new int[sz]), size(sz) {
        for(int i = 0; i < sz; i++) {
            array[i] = i;
        }
    }

    int at(int i) {
        assert(i < size);
        return array[i];
    }
private:
    int *array;
    int size;
};
```

Destructors

```
#include <iostream>
#include "sequence.h"

using std::cout;  using std::endl;

int main() {
    Sequence seq(10);
    for(int i = 0; i < 10; i++) {
        cout << seq.at(i) << ' ';
    }
    cout << endl;
    return 0;
}
```

```
$ g++ -c sequence_main.cpp -std=c++11 -pedantic -Wall -Wextra -g
$ g++ -o sequence_main sequence_main.o
$ ./sequence_main
0 1 2 3 4 5 6 7 8 9
```

Destructors

```
$ valgrind --leak-check=full ./sequence_main
0 1 2 3 4 5 6 7 8 9
==26== Memcheck, a memory error detector
==26== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==26== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==26== Command: ./sequence_main
==26==
==26==
==26== HEAP SUMMARY:
==26==     in use at exit: 40 bytes in 1 blocks
==26==   total heap usage: 3 allocs, 2 frees, 76,840 bytes allocated
==26==
==26== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==26==    at 0x4C308B7: operator new[](unsigned long) (vg_replace_malloc.c:423)
==26==    by 0x4009D4: Sequence::Sequence(int) (sequence.h:6)
==26==    by 0x4008FF: main (sequence_main.cpp:7)
==26==
==26== LEAK SUMMARY:
==26==    definitely lost: 40 bytes in 1 blocks
==26==    indirectly lost: 0 bytes in 0 blocks
==26==    possibly lost: 0 bytes in 0 blocks
==26==    still reachable: 0 bytes in 0 blocks
==26==          suppressed: 0 bytes in 0 blocks
==26==
==26== For counts of detected and suppressed errors, rerun with: -v
==26== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Destructors

Allocates new `int[sz]` in constructor, but never `delete[]`s it

It's common for a constructor to obtain a resource (allocate memory, open a file, etc) that should be released when object is destroyed

Destructor is a function called by C++ when the object goes out of scope or is otherwise deallocated (i.e. with `delete`)

Destructors

```
#include <cassert>

class Sequence {
public:
    Sequence(int sz) : array(new int[sz]), size(sz) {
        for(int i = 0; i < sz; i++) {
            array[i] = i;
        }
    }

    // *** Destructor (name starts with tilde) ****
    ~Sequence() { delete[] array; }

    int at(int i) {
        assert(i < size);
        return array[i];
    }
private:
    int *array;
    int size;
};

};
```

Destructors

```
$ g++ -c sequence_main.cpp -std=c++11 -pedantic -Wall -Wextra -g  
$ g++ -o sequence_main sequence_main.o  
$ ./sequence_main  
0 1 2 3 4 5 6 7 8 9
```



Destructors

```
$ valgrind --leak-check=full ./sequence_main
0 1 2 3 4 5 6 7 8 9
==34== Memcheck, a memory error detector
==34== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==34== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==34== Command: ./sequence_main
==34==
==34==
==34== HEAP SUMMARY:
==34==     in use at exit: 0 bytes in 0 blocks
==34==   total heap usage: 3 allocs, 3 frees, 76,840 bytes allocated
==34==
==34== All heap blocks were freed -- no leaks are possible
==34==
==34== For counts of detected and suppressed errors, rerun with: -v
==34== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Destructors

Destructors are better than having a special member function for releasing resources; e.g.:

```
#include <cassert>

class Sequence {
public:
    ...
    // User must call clean_up when finished with Sequence
    void clean_up() { delete[] array; }
    ...
};
```

Destructors

Here, user forgets to call clean_up:

```
{  
    Sequence s(40);  
    // ... (no call to s.clean_up())  
} // s goes out of scope and memory is leaked
```

More subtly:

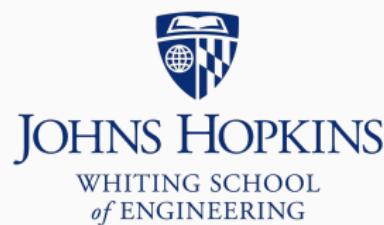
```
{  
    Sequence s(40);  
    if(some_condition) {  
        return 0; // memory leaked!  
    }  
    s.clean_up();  
}
```

Passing by reference

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Passing by reference

We've seen pass-by-reference versus pass-by-value

In C++, when passing objects (class or struct variables), we usually choose to pass by reference

- const reference if modification is not permitted
- Normal reference otherwise

Passing by reference

What's the difference?

```
int sum(vector<int> vec) { ... };
```

```
int sum(const vector<int>& vec) { ... };
```

Passing by reference

```
// Creates a copy of vec  
int sum(vector<int> vec) { ... };  
  
// *Does not* create a copy of vec  
int sum(const vector<int>& vec) { ... };
```

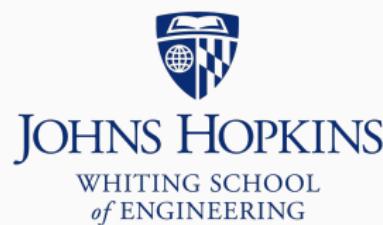
Second form avoids making a (potentially expensive) copy

We also pass by reference for *dynamic binding*, as we'll discuss later

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Inheritance

C++ classes can be related to each other

```
class Account {...}; class CheckingAccount {...};
```

- “is a” relationship; a checking account is a kind of account

```
class GradeList {...}; vector<double>
```

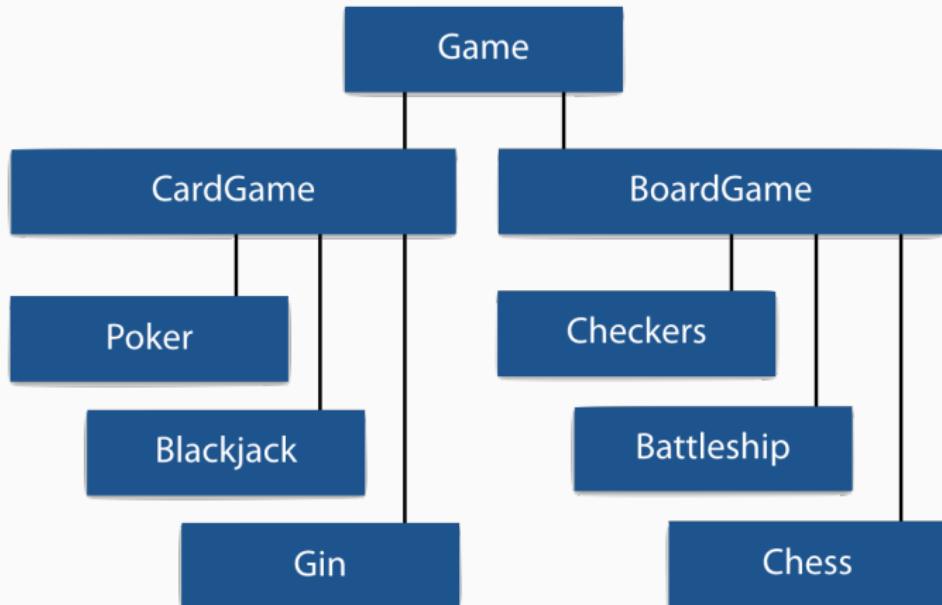
- “has a” relationship; grade list *has* vector of grades as a field

Inheritance

Base class	Derived class
Account	CheckingAccount, SavingsAccount
Shape	Rectangle, Circle
Document	WordDoc, WebPage, TextDoc

Example “is a” relationships

Inheritance



Multiple levels of “is a” relationships

Inheritance

Derived class inherits from *base class*

Java-like vocab: *subclass* inherits from *superclass*

(We'll typically say “derived” and “base”)

Inheritance

```
class BaseClass {  
    // Definitions for BaseClass  
    ...  
};  
  
class DerivedClass: public BaseClass {  
    // Definitions for DerivedClass  
    ...  
};
```

This is “public inheritance” – by far the most common kind
(protected & private inheritance also possible, but rarely used)

Inheritance

Derived class *inherits all members* of base class, whether public, protected or private, except:

- Constructors
- Assignment operator (discussed later)

Derived class cannot delete things it inherited; cannot pick and choose what to inherit

- But derived class can *override* inherited member functions
- override = substitute own implementation for base class's

Inheritance

Base-class members marked public or protected can be accessed from member functions defined in the derived class

Base-class members marked private *cannot* be accessed from member functions defined in the derived class

- They're still there, and *base class* member functions can still use them, but *derived class* member functions can't

Inheritance

protected is an access modifier we haven't used yet

- protected fields & functions cannot be accessed except from member functions of class (like private)
- They are accessible from member functions defined in derived classes (like public)

Inheritance

```
class Account {  
public:  
    Account() : balance(0.0) {}  
    Account(double initial) : balance(initial) {}  
  
    void credit(double amt) { balance += amt; }  
    void debit(double amt) { balance -= amt; }  
    double get_balance() const { return balance; }  
  
private:  
    double balance;  
};
```

Default constructor sets balance to 0; non-default constructor sets according to argument

balance is private, modified via credit(amt)/debit(amt)

Inheritance

What does this const mean?

```
class Account {  
public:  
    ...  
    double get_balance() const { return balance; }  
    //          ^^^^^^  
private:  
    double balance;  
};
```

Means member function *does not modify any fields*

- `get_balance()` does not modify `balance`

If you have a `const Account` (or `const Account &`), `const` member functions are the *only* ones you can call

Inheritance

```
#include <iostream>
#include "account.h"

using std::cout;  using std::endl;

int main() {
    Account acct(1000.0);
    acct.credit(1000.0);
    acct.debit(100.0);
    cout << "Balance is: " << acct.get_balance() << endl;
    return 0;
}

$ g++ -c account_main1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o account_main1 account_main1.o
$ ./account_main1
Balance is: 1900
```

Inheritance

```
class CheckingAccount : public Account {  
public:  
    CheckingAccount(double initial, double atm) :  
        Account(initial), total_fees(0.0), atm_fee(atm) {}  
  
    void cash_withdrawal(double amt) {  
        total_fees += atm_fee;  
        debit(amt + atm_fee);  
    }  
  
    double get_total_fees() const { return total_fees; }  
  
private:  
    double total_fees;  
    double atm_fee;  
};
```

Inheritance

```
class SavingsAccount : public Account {  
public:  
    SavingsAccount(double initial, double rate) :  
        Account(initial), annual_rate(rate) {}  
  
    // Not implemented here; usual compound interest calc  
    double total_after_years(int years) const;  
  
private:  
    double annual_rate;  
};
```

Inheritance

Syntax for declaring a class that derives from another:

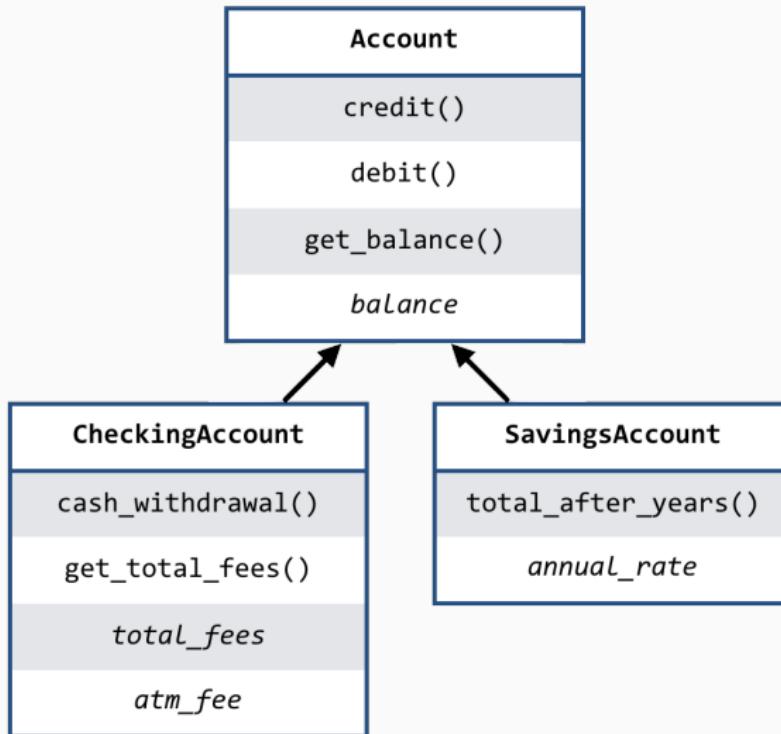
```
class Derived : public Base {  
    ...  
};
```

Who can use members with public, protected and private access modifiers?

Access modifier	Any function	Derived-class members	Same-class members
public	Yes	Yes	Yes
protected	No	Yes	Yes
private	No	No	Yes

Inheritance

Returning to our financial account example:



Inheritance

```
#include <iostream>
#include "account.h"
using std::cout;  using std::endl;

int main() {
    Account acct(1000.0);
    acct.credit(1000.0);
    acct.debit(100.0);
    cout << "Account balance is: $" << acct.get_balance() << endl;

    CheckingAccount checking(1000.0, 2.00);
    checking.credit(1000.0);
    checking.cash_withdrawal(100.0); // incurs ATM fee
    cout << "Checking balance is: $" << checking.get_balance() << endl;
    cout << "Checking total fees is: $" << checking.get_total_fees() << endl;

    SavingsAccount saving(1000.0, 0.05);
    saving.credit(1000.0);
    cout << "Savings balance is: $" << saving.get_balance() << endl;
    return 0;
}
```

Inheritance

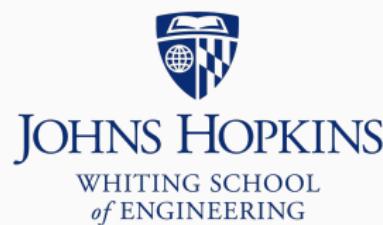
```
$ g++ -c account_main2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o account_main2 account_main2.o
$ ./account_main2
Account balance is: $1900
Checking balance is: $1898
Checking total fees is: $2
Savings balance is: $2000
```

Polymorphism

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Polymorphism

```
class Account {  
public:  
    ...  
    std::string type() const { return "Account"; }  
    ...  
};  
  
class CheckingAccount : public Account {  
public:  
    ...  
    std::string type() const { return "CheckingAccount"; }  
    ...  
};  
  
class SavingsAccount : public Account {  
public:  
    ...  
    std::string type() const { return "SavingsAccount"; }  
    ...  
};
```

Polymorphism

```
#include <iostream>
#include "account2.h"

using std::cout;  using std::endl;

void print_account_type(const Account& acct) {
    cout << acct.type() << endl;
}

int main() {
    Account acct(1000.0);
    CheckingAccount checking(1000.0, 2.00);
    SavingsAccount saving(1000.0, 0.05);

    print_account_type(acct);
    print_account_type(checking);
    print_account_type(saving);

    return 0;
}
```

Polymorphism

Note the types:

```
void print_account_type(const Account& acct) {
    cout << acct.type() << endl;
}

int main() {
    ...
    CheckingAccount checking(1000.0, 2.00);
    ...
    print_account_type(checking);
    ...
}
```

Polymorphism

In main, checking_acct has type CheckingAccount

Passed to print_account_type as const Account&; you may use
a variable of a derived type *as though it has the base type*

- Makes sense: CheckingAccount *is a* Account

Polymorphism

```
int main() {
    vector<Account> my_accounts;

    // this is OK; CheckingAccount is
    // derived from Account
    my_accounts.push_back(CheckingAccount(2000.0));

    cout << my_accounts.back().type() << endl;
    return 0;
}
```

Polymorphism

```
void print_account_type(const Account& acct) {
    cout << acct.type() << endl;
}

int main() {
    ...
    CheckingAccount checking(1000.0, 2.00);
    ...
    print_account_type(checking_acct);
    ...
}
```

Does `acct.type()` call:

- `Account::type()` – the parameter's type?
- `CheckingAccount::type()` – checking's declared type?

Polymorphism

```
$ g++ -c account_main3.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o account_main3 account_main3.o  
$ ./account_main3  
Account  
Account  
Account
```

It calls `Account::type()`

What if we want the call the member corresponding to the *original declared type* of the variable (`CheckingAccount`) rather than the base type we happen to be using for it right now (`Account`)?

This requires *dynamic binding*

- Declare relevant member functions as *virtual*:

Polymorphism

```
class Account {  
public:  
    ...  
    virtual std::string type() const { return "Account"; }  
    ...  
};  
  
class CheckingAccount : public Account {  
public:  
    ...  
    virtual std::string type() const { return "CheckingAccount"; }  
    ...  
};  
  
class SavingsAccount : public Account {  
public:  
    ...  
    virtual std::string type() const { return "SavingsAccount"; }  
    ...  
};
```

Polymorphism

```
#include <iostream>
#include "account3.h" // now with *virtual* `type` member functions

using std::cout; using std::endl;

void print_account_type(const Account& acct) {
    cout << acct.type() << endl;
}

int main() {
    Account acct(1000.0);
    CheckingAccount checking(1000.0, 2.00);
    SavingsAccount saving(1000.0, 0.05);

    print_account_type(acct);
    print_account_type(checking);
    print_account_type(saving);

    return 0;
}
```

Polymorphism

```
$ g++ -c account_main4.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o account_main4 account_main4.o  
$ ./account_main4  
Account  
CheckingAccount  
SavingsAccount
```

Dynamic binding / virtual functions enable C++ *polymorphism*

- In Java & Python *all methods are virtual*; you have no choice

A class object can look “on the surface” like the base class while behaving like the derived class

Polymorphism

```
#include <iostream>
#include "account3.h" // still with *virtual* `type` member functions

using std::cout; using std::endl;

// !!! *** Pass by value this time *** !!!
void print_account_type(Account acct) {
    cout << acct.type() << endl;
}

int main() {
    Account acct(1000.0);
    CheckingAccount checking(1000.0, 2.00);
    SavingsAccount saving(1000.0, 0.05);
    print_account_type(acct);
    print_account_type(checking);
    print_account_type(saving);
    return 0;
}
```

Polymorphism

```
$ g++ -c account_main5.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o account_main5 account_main5.o  
$ ./account_main5  
Account  
Account  
Account
```

Fields only in the derived class are simply not copied when passed by value using the base type

- This is *object slicing*

Passing by reference doesn't cause slicing and preserves our ability to call virtual functions in the derived class; another reason to prefer passing class variables by reference

Polymorphism



<https://www.completelydelicious.com/cut-cake-even-layers/>

Polymorphism

```
class Base {           class Derived : public Base {  
public:                 public:  
    void      normal();     void      normal();  
    virtual void virt();    virtual void virt();  
};                      };
```

Say we declare `Derived o` and later pass `o` to a function. The member functions called with `.normal()` and `.virt()` depend on the parameter type.

Param type	a.normal()	a.virt()	What happens?
Derived& a	Derived::normal()	Derived::virt()	Passed by ref
Derived a	Derived::normal()	Derived::virt()	Copied, not sliced
Base& a	Base::normal()	Derived::virt()	Passed by ref
Base a	Base::normal()	Base::virt()	Sliced & copied

Polymorphism

Same reasoning applies when converting between related types implicitly or using casting:

```
int main() {
    Derived original;

    // sliced; b.virt() -> Base::virt()
    Base b          = (Base)original;

    // reference; bref.virt() -> Derived::virt()
    Base& bref      = original;

    Derived d       = original; // simple copy
    Derived& dref  = original; // simple reference

    return 0;
}
```

Polymorphism

Virtual functions allow a class variable to remember and act like its declared type even when temporarily taking the base type

- ... but *how* do they remember?

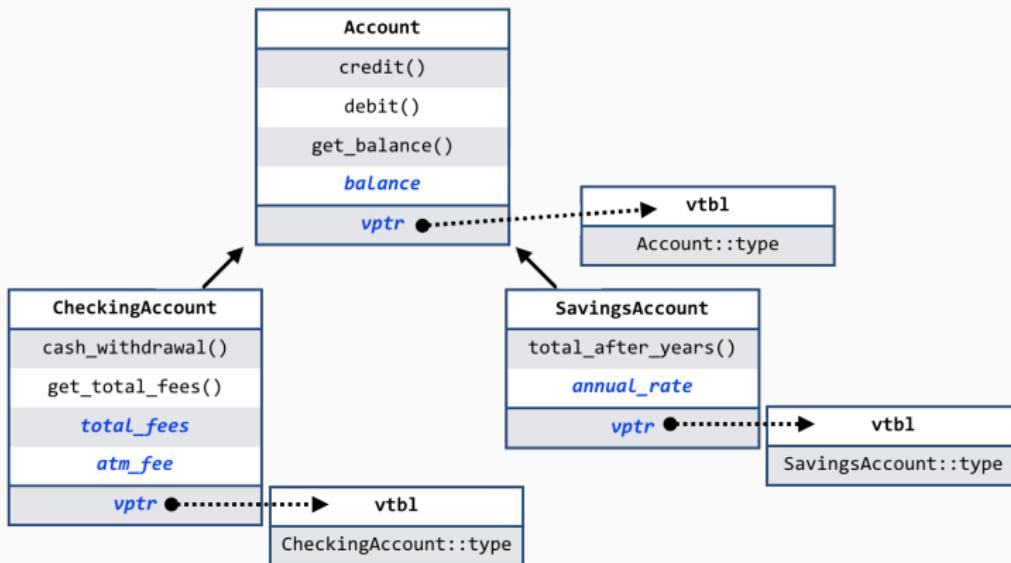
When a class has a virtual member function, it also gets a hidden *virtual function table* or *vtable*

- Points to declared-type implementation

Table adds 1 pointer (8 bytes) to size of class, regardless of number of virtual functions

Polymorphism

Blue-highlighted items take memory in class variables. Memory is also needed for function code and virtual-function tables (vtbls), but those aren't per-object, and `sizeof` won't include them.



Polymorphism

Sometimes the base class is too generic to be useful as much more than a “guide” for derived classes

```
class Shape {  
public:  
    virtual double area() const { } // NO SENSIBLE IMPLEMENTATION FOR THIS  
};  
  
class Rectangle : public Shape {  
public:  
    ...  
    virtual double area() const { return width * height; }  
    ...  
private:  
    double width, height;  
};  
  
class Circle : public Shape {  
public:  
    ...  
    virtual double area() const { return PI * radius * radius; }  
    ...  
private:  
    double radius;  
};
```

Polymorphism

We use *pure virtual* functions to flag member functions that *cannot* be usefully implemented in the base class and *must* be overridden and implemented in a derived class

```
class Shape {  
public:  
    virtual double area() const = 0; // pure virtual  
};
```

A class with any *pure virtual* functions is an *abstract base class*

- You may not instantiate an abstract base class
- E.g. Shape s; is not allowed; use derived type(s) instead

Polymorphism

```
#include <iostream>

using std::cout;  using std::endl;

class Shape {
public:
    virtual double area() const = 0;
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : width(w), height(h) { }
    virtual double area() const { return width * height; }
private:
    double width, height;
};

int main() {
    Shape s;
    Rectangle r = {10.0, 5.0};
    cout << "r area = " << r.area() << endl;
    return 0;
}
```

Polymorphism

```
$ g++ -c shapes.cpp -std=c++11 -pedantic -Wall -Wextra
shapes.cpp: In function 'int main()':
shapes.cpp:19:11: error: cannot declare variable 's' to be of abstract type
'Shape'
    Shape s;
               ^
shapes.cpp:5:7: note:  because the following virtual functions are pure within
'Shape':
class Shape {
    ~~~~~
shapes.cpp:7:20: note:  virtual double Shape::area() const
    virtual double area() const = 0;
               ^~~~~
```

Polymorphism

```
#include <iostream>

using std::cout;  using std::endl;

class Shape {
public:
    virtual double area() const = 0;
};

class Rectangle : public Shape {
public:
    Rectangle(double w, double h) : width(w), height(h) { }
    virtual double area() const { return width * height; }
private:
    double width, height;
};

int main() {
    // Shape s; // **** got rid of this ****
    Rectangle r = {10.0, 5.0};
    cout << "r area = " << r.area() << endl;
    return 0;
}
```

Polymorphism

```
$ g++ -c shapes.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o shapes shapes.o  
$ ./shapes  
r area = 50
```

Polymorphism

virtual syntax details:

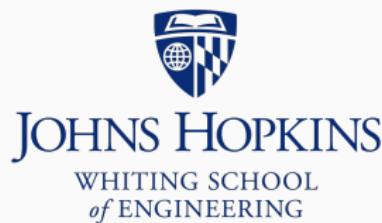
- Omit virtual keyword when defining a virtual member function outside a class definition
- When declaring a virtual member function that overrides one in the parent class, you *may* omit the virtual keyword
- When you declare a virtual member function that should override one in the base class, use the override keyword
 - Helps catch mistakes where you intended to override but failed to due to a minor difference in function signature

Virtual destructors

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Virtual destructors

```
class Base {
public:
    Base() : base_memory(new char[1000]) { }

    ~Base() { delete[] base_memory; }

private:
    char *base_memory;
};

class Derived : public Base {
public:
    Derived() : Base(), derived_memory(new char[1000]) { }

    ~Derived() { delete[] derived_memory; }

private:
    char *derived_memory;
};
```

Virtual destructors

```
#include "virt_dtor.h"

int main() {
    // Note use of base-class pointer
    Base *obj = new Derived();
    delete obj; // calls what destructor(s)?
    return 0;
}
```

new Derived() calls Derived default constructor, which in turn calls Base default constructor; that's good

Which destructor is called?

- Destructor is not virtual
- Does that mean ~Base is called but not ~Derived?

Virtual destructors

```
$ g++ -o virt_dtor virt_dtor.cpp
$ valgrind ./virt_dtor
==22== Memcheck, a memory error detector
==22== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==22== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==22== Command: ./virt_dtor
==22==
==22==
==22== HEAP SUMMARY:
==22==     in use at exit: 1,000 bytes in 1 blocks
==22==   total heap usage: 4 allocs, 3 frees, 74,720 bytes allocated
==22==
==22== LEAK SUMMARY:
==22==     definitely lost: 1,000 bytes in 1 blocks
==22==     indirectly lost: 0 bytes in 0 blocks
==22==     possibly lost: 0 bytes in 0 blocks
==22==     still reachable: 0 bytes in 0 blocks
==22==           suppressed: 0 bytes in 0 blocks
==22== Rerun with --leak-check=full to see details of leaked memory
==22==
==22== For counts of detected and suppressed errors, rerun with: -v
==22== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

~Derived is *not* called; derived_memory is leaked

Virtual destructors

```
class Base {
public:
    Base() : base_memory(new char[1000]) { }

    // Now *** virtual ***
    virtual ~Base() { delete[] base_memory; }

private:
    char *base_memory;
};

class Derived : public Base {
public:
    Derived() : Base(), derived_memory(new char[1000]) { }

    // Now *** virtual ***
    virtual ~Derived() { delete[] derived_memory; }

private:
    char *derived_memory;
};
```

Virtual destructors

```
#include "virt_dtor2.h"

int main() {
    // Note use of base-class pointer
    Base *obj = new Derived();
    delete obj; // calls what destructor(s)?
    return 0;
}
```

Virtual destructors

```
$ g++ -o virt_dtor2 virt_dtor2.cpp
$ valgrind ./virt_dtor2
==28== Memcheck, a memory error detector
==28== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==28== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==28== Command: ./virt_dtor2
==28==
==28==
==28== HEAP SUMMARY:
==28==     in use at exit: 0 bytes in 0 blocks
==28==   total heap usage: 4 allocs, 4 frees, 74,728 bytes allocated
==28==
==28== All heap blocks were freed -- no leaks are possible
==28==
==28== For counts of detected and suppressed errors, rerun with: -v
==28== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fixed; thanks to dynamic binding, delete obj calls ~Derived,
which in turn calls ~Base

Derived-class destructor always implicitly calls base-class destructor
at the end

Virtual destructors

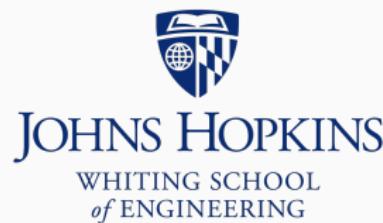
To avoid this in general: *Any class with virtual member functions* should also have a virtual destructor, even if the destructor does nothing

Overloading & operator overloading

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Overloading & operator overloading

C++ compiler can distinguish functions with same name but different parameters

```
#include <iostream>

using std::cout;  using std::endl;

void output_type(int)  { cout << "int" << endl; }
void output_type(float) { cout << "float" << endl; }

int main() {
    output_type(1);    // int argument
    output_type(1.f); // float argument
    return 0;
}
```

Overloading

```
$ g++ -c print_type.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o print_type print_type.o  
$ ./print_type  
int  
float
```

But it *cannot* distinguish functions with same name & parameters
but different return types

Overloading

```
#include <iostream>

using std::cout;  using std::endl;

int    get_one() { return 1; }
float get_one() { return 1.0f; }

int main() {
    int i = get_one();
    float f = get_one();
    cout << i << ' ' << f << endl;
    return 0;
}
```

Overloading

```
$ g++ -c print_type.cpp -std=c++11 -pedantic -Wall -Wextra
print_type.cpp: In function 'float get_one()':
print_type.cpp:6:7: error: ambiguating new declaration of 'float get_one()'
    float get_one() { return 1.0f; }
          ^~~~~~
print_type.cpp:5:7: note: old declaration 'int get_one()'
    int    get_one() { return 1; }
          ^~~~~~
```

Operator overloading

Operators like + and << are like functions

a + b is like plus(a, b) or a.plus(b)

a + b + c is like plus(plus(a, b), c)

Operator overloading

classes override member functions to customize their behavior

Operator overloading is when we do something similar for *operators*

We've been using it:

```
string msg("Hello"), name;  
cin >> name; // >> is an operator  
cout << msg << ", "; // << is an operator  
cout << (name + '!') << endl; // << and + are operators
```

Operator overloading

`cout <<` works with many types, but not all:

```
#include <iostream>
#include <vector>

using std::cout;  using std::endl;
using std::vector;

int main() {
    vector<int> vec = {1, 2, 3};
    cout << vec << endl;
    return 0;
}
```

Operator overloading

```
$ g++ -c insertion_eg1.cpp -std=c++11 -pedantic -Wall -Wextra
insertion_eg1.cpp: In function 'int main()':
insertion_eg1.cpp:9:10: error: no match for 'operator<<' (operand types are
'std::ostream {aka std::basic_ostream<char>}' and 'std::vector<int>')
    cout << vec << endl;
    ~~~~~^~~~~~
In file included from /usr/include/c++/7/iostream:39:0,
                 from insertion_eg1.cpp:1:
/usr/include/c++/7/ostream:108:7: note: candidate: std::basic_ostream<_CharT,
      _Traits>::__ostream_type& std::basic_ostream<_CharT,
      _Traits>::operator<<(std::basic_ostream<_CharT, _Traits>::__ostream_type&
(*) (std::basic_ostream<_CharT, _Traits>::__ostream_type&)) [with _CharT = char,
      _Traits = std::char_traits<char>; std::basic_ostream<_CharT,
      _Traits>::__ostream_type = std::basic_ostream<char>]
          operator<<(__ostream_type& (*__pf)(__ostream_type&))
          ~~~~~^~~~~~
/usr/include/c++/7/ostream:108:7: note:   no known conversion for argument 1
from 'std::vector<int>' to 'std::basic_ostream<char>::__ostream_type&
(*) (std::basic_ostream<char>::__ostream_type&) {aka std::basic_ostream<char>&
(*) (std::basic_ostream<char>&)}'
/usr/include/c++/7/ostream:117:7: note: candidate: std::basic_ostream<_CharT,
      _Traits>::__ostream_type& std::basic_ostream<_CharT,
      _Traits>::operator<<(std::basic_ostream<_CharT, _Traits>::__ios_type&
(*) (std::basic_ostream<_CharT, _Traits>::__ios_type&)) [with _CharT = char,
      _Traits = std::char_traits<char>; std::basic_ostream<_CharT,
      _Traits>::__ostream_type = std::basic_ostream<char>; std::basic_ostream<_CharT,
      _Traits>::__ios_type = std::basic_ios<char>]
          operator<<(__ios_type& (*__pf)(__ios_type&))
          ~~~~~^~~~~~
```

Operator overloading

We can make it work by defining the appropriate operator overload:

```
#include <iostream>
#include <vector>

using std::cout;  using std::endl;
using std::vector;

std::ostream& operator<<(std::ostream& os, const vector<int>& vec) {
    for(vector<int>::const_iterator it = vec.cbegin();
        it != vec.cend(); ++it)
    {
        os << *it << ' ';
    }
    return os;
}

int main() {
    const vector<int> vec = {1, 2, 3};
    cout << vec << endl; // now this will work!
    return 0;
}
```

Operator overloading

```
$ g++ -c insertion_eg2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o insertion_eg2 insertion_eg2.o  
$ ./insertion_eg2  
1 2 3
```

Operator overloading

`std::ostream` is a C++ *output stream*

Can write to it, can't read from it

It is `cout`'s type

- `cout` can be passed as parameter of type `std::ostream& os`
- `const std::ostream&` won't work, since it disallows writing

Operator overloading

When we see this:

```
cout << "Hello " << 1 << ' ' << 2;
```

This is what really happens:

```
operator<<(  
    operator<<(  
        operator<<(  
            operator<<(cout, "Hello"), 1), ' '), 2)
```

- First, we cout << "Hello"
- That *returns* cout, which becomes the new left operand
- Then we cout << 1, returning cout as new left operand
- Then we cout << ' ', returning cout
- ...

Operator overloading

```
std::ostream& operator<<(std::ostream& os, const vector<int>& vec) {  
    for(vector<int>::const_iterator it = vec.cbegin();  
        it != vec.cend(); ++it)  
    {  
        os << *it << ' ';  
    }  
    return os;  
}
```

Allows `vector<int>` to appear in a typical `cout <<` chain

- Taking `std::ostream& os` in 1st parameter & returning `os` enables chaining
- Taking `const vector<int>&` as 2nd parameter allows `vector<int>` to appear as right operand in `operator<<` call
- Passing by `const` reference avoids needless copying

Operator overloading

Can also overload operator with a member function

```
class Complex {  
public:  
    Complex(double r, double i) : real(r), imaginary(i) {}  
  
    Complex operator+(const Complex& rhs) const {  
        // left operand is the current object ("myself")  
        // right operand is rhs  
        // return value is a new Complex  
        Complex result(real + rhs.real, imaginary + rhs.imaginary);  
        return result; // Note: return type can't be `Complex&`  
    }  
  
private:  
    double real, imaginary;  
};
```

Operator overloading

Say we want to be able to print a Complex with cout <<

```
class Complex {  
public:  
    ...  
private:  
    double real, imaginary;  
};
```

Would this work?

```
std::ostream& operator<<(std::ostream& os, const Complex& d) {  
    os << d.real << " + " << d.imaginary + "i";  
    return os;  
}
```

Operator overloading

No, because `real` & `imaginary` are private

Can we make `operator<<` be a member function for `Complex`?

No; `operator<<` takes `std::ostream` as left operand, but a member function has to take the *object itself* as left operand

Operator overloading

Can't be a member, but also needs access to private members??

Solution: a friend function:

```
class Complex {  
public:  
    ...  
  
    friend std::ostream& operator<<(std::ostream& os, Complex c);  
  
private:  
    double real, imaginary;  
};  
  
std::ostream& operator<<(std::ostream& os, Complex c) {  
    os << c.real << " + " << c.imaginary << 'i';  
    return os;  
}
```

Operator overloading

operator<< is a separate, non-member function

It is also a friend of Complex, as declared here:

```
class Complex {  
public:  
    ...  
    friend std::ostream& operator<<(std::ostream& os, Complex c);  
    ...  
};
```

A friend of a class can access its private members

- Like a “backstage pass”

Operator overloading

Access modifier	Any function	Derived-class members	Same-class members & friends
public	Yes	Yes	Yes
protected	No	Yes	Yes
private	No	No	Yes

Often, friend is to be avoided, and signals bad design

A common exception is when operator<< must access private fields

Operator overloading

When using a member function to overload an operator, sometimes the return value is “myself”

E.g. consider the `+=` compound operator

```
int c = 0;  
c += (c += 2);
```

Recall that the result of an assignment (including compound assignment) is the value assigned

Here, `(c += 2)` both sets `c` to 2 and evaluates to 2

Then outer `c += (...)` sets `c` to 4

Operator overloading

Now for Complex:

```
Complex c(0.0, 0.0);
c += (c += Complex(2.0, 2.0));
```

Operator overloading

Can return “myself” by returning `*this` from operator`+=` as below

Member functions have implicit pointer argument `this`: a pointer to “myself,” the current object

```
class Complex {  
public:  
    ...  
    Complex operator+=(const Complex& rhs) {  
        real += rhs.real;  
        imaginary += rhs.imaginary;  
        return *this;  
    }  
    ...  
};
```

Operator overloading

Operators we can overload:

+	-	*	/	%	^	&		~
!	->	=	<	>	<=	>=	++	--
<<	>>	==	!=	&&		+=	--	/=
%=	^=	&=	=	*=	<<=	>>=	[]	()

These are common:

- +, -, +=, -= (e.g. Complex)
- *, -> (e.g. iterators)
- [] (e.g. vector)
- = (most things that can be copied)
- ==, <, >, <=, >= (things that can be compared)

Operator overloading

Overloading allowed only when one or both operands are a user-defined type, like a class

This won't work (thank goodness):

```
int operator+(int a, int b) {  
    return a - b;  
}
```

Can't change operator's precedence, associativity, or # operands

Operator overloading

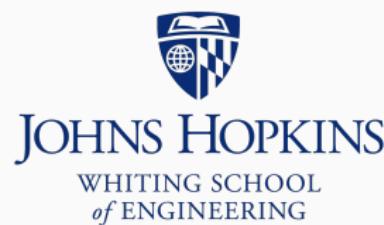
To allow Date to work with std::sort, we only have to add operator<:

```
class Date {  
public:  
    ...  
    bool operator<(const Date& rhs) const {  
        if(year < rhs.year) return true;  
        if(year > rhs.year) return false;  
        if(month < rhs.month) return true;  
        if(month > rhs.month) return false;  
        return day < rhs.day;  
    }  
    ...  
};
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

enum class

We have often used integers to describe *categorical* data

```
#include <stdlib.h>

int main() {
    void *memory = malloc(1000000);
    if(memory == NULL) {
        return 1; // failure
    }
    // do something with memory
    return 0; // success
}
```

Returning 0 means “success”, 1 means “failure”

enum class

```
struct Card {  
    int rank; // 1=ace, 2=two, ..., 10=ten  
              // 11=jack, 12=queen, 13=king  
    int suit; // 0=heart, 1=club, 2=diamond, 3=spade  
  
    Card(int r, int s) : rank(r), suit(s) {}  
};
```

int has advantages; e.g. we can compare ranks with <

Also has disadvantages:

- Mapping between ints and suits is arbitrary
- If we mix up rank and suit – e.g. Card c(3, 13) – compiler can't catch it

`enum class`

`enum class` creates a *categorical* type

(C & C++ have an older mechanism called simply `enum` that we won't discuss here)

enum class

```
#include <iostream>

using std::cout;  using std::endl;

enum class Suit {
    HEART, CLUB, DIAMOND, SPADE
};

struct Card {
    int rank; // 1=ace, 11=jack, 12=queen, 13=king
    Suit suit;

    Card(int r, Suit s) : rank(r), suit(s) { }

};

int main() {
    Card c(1, Suit::CLUB); // ace of clubs
    cout << "c.suit = " << (int)c.suit << endl;
    return 0;
}
```

enum class

```
$ g++ -c enum_1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o enum_1 enum_1.o  
$ ./enum_1  
c.suit = 1
```

Behind the scenes, an enum class is really an int

- Starts at 0, so HEART=0, CLUB=1, DIAMOND=2, SPADE=3

C++ will refuse to implicitly convert between enum class and int;
we had to explicitly cast for cout

```
cout << "c.suit = " << (int)c.suit << endl;
```

enum class

```
// *** cards.h ***

enum class Suit { HEART, CLUB, DIAMOND, SPADE };

enum class Rank {
    // "= 1" to start numbering at 1 instead of 0
    ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
    EIGHT, NINE, TEN, JACK, QUEEN, KING
};

struct Card {
    Rank rank;
    Suit suit;

    Card(Rank r, Suit s) : rank(r), suit(s) { }

};
```

enum class

```
// *** cards.cpp ***

#include <iostream>
#include "cards.h"

using std::cout;  using std::endl;

int main() {
    Card c1(Rank::SEVEN, Suit::HEART);
    Card c2((Rank)10, Suit::DIAMOND);
    cout << "c1=" << (int)c1.rank << ", s=" << (int)c1.suit << endl;
    cout << "c2=" << (int)c2.rank << ", s=" << (int)c2.suit << endl;
    return 0;
}

$ g++ -c cards.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o cards cards.o
$ ./cards
c1=7, s=0
c2=10, s=2
```

enum class

We get a compiler error if we mix up rank & suit

```
#include <iostream>
#include "cards.h"

using std::cout;  using std::endl;

int main() {
    Card c1(Suit::HEART, Rank::SEVEN); // oops!
    cout << "c1=" << (int)c1.rank << ", s=" << (int)c1.suit << endl;
    return 0;
}
```

enum class

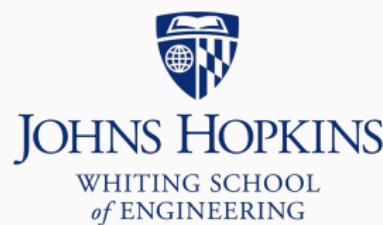
```
$ g++ -c cards_error.cpp -std=c++11 -pedantic -Wall -Wextra
cards_error.cpp: In function 'int main()':
cards_error.cpp:7:37: error: no matching function for call to 'Card::Card(Suit,
Rank)'
    Card c1(Suit::HEART, Rank::SEVEN); // oops!
                                         ^
In file included from cards_error.cpp:2:0:
cards.h:15:5: note: candidate: Card::Card(Rank, Suit)
    Card(Rank r, Suit s) : rank(r), suit(s) { }
                           ^
cards.h:15:5: note:   no known conversion for argument 1 from 'Suit' to 'Rank'
cards.h:11:8: note: candidate: constexpr Card::Card(const Card&)
struct Card {
    ^
cards.h:11:8: note:   candidate expects 1 argument, 2 provided
cards.h:11:8: note: candidate: constexpr Card::Card(Card&&)
cards.h:11:8: note:   candidate expects 1 argument, 2 provided
```

static class members

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

static class members

A class member marked static behaves like a “global” variable or function

- Typical members are associated with an *instance* of the class; a field takes up space in every class variable
- static members are associated with the class, but not with instances
- this pointer is not available in static member functions

static class members

```
class Temperature {
public:
    Temperature(double f) : fahrenheit(f) { }

    static double fahrenheit_to_celcius(double f) {
        return 5.0 / 9.0 * (f - 32);
    }

    static double celcius_to_kelvin(double c) { return c + 273.15; }

    static float fahrenheit_to_kelvin(double f) {
        return celcius_to_kelvin(fahrenheit_to_celcius(f));
    }

    double as_fahrenheit() const { return fahrenheit; }
    double as_celcius() const { return fahrenheit_to_celcius(fahrenheit); }
    double as_kelvin() const { return fahrenheit_to_kelvin(fahrenheit); }

    static const int FREEZING = 32;

    bool is_freezing() const { return fahrenheit <= FREEZING; }

private:
    double fahrenheit;
};
```

static class members

Static

- fahrenheit_to_celcius
- celcius_to_kelvin
- fahrenheit_to_kelvin
- FREEZING
 - Stored once & doesn't take space in Temperature variables

Non-static

- Constructor
- as_fahrenheit, as_celcius, as_kelvin
- is_freezing
- fahrenheit
 - Only this takes space in Temperature variables

static class members

```
#include <iostream>
#include <cassert>
#include "temp.h"

int main() {
    Temperature t1(30.0);
    assert(t1.is_freezing());      // non-static
    assert(t1.as_celcius() < 0); // non-static
    assert(Temperature::fahrenheit_to_celcius(33) > 0); // static
    std::cout << "Assertions passed" << std::endl;
    return 0;
}
```

```
$ g++ -c temp.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o temp temp.o
$ ./temp
Assertions passed
```

static class members

From outside the class member functions, use A::B to access static member B of class A

- `Temperature::fahrenheit_to_celcius(33)`

`static const` fields with integer type can be initialized immediately:

- `static const int FREEZING = 32`
- Typical convention is to capitalize `static const` fields

Otherwise, they are declared in the class definition but defined later in a `.cpp` file

static class members

```
// *** circle.h ***
class Circle {
public:
    static const float PI;
    Circle(double r) : radius(r) { }
    double area() const { return PI * radius * radius; }

private:
    double radius;
};

// *** circle.cpp ***
#include "circle.h"

const float Circle::PI = 3.14f;
```

static class members

```
// *** circle_main.cpp ***

#include <iostream>
#include "circle.h"

int main() {
    std::cout << "PI is " << Circle::PI << std::endl;
    return 0;
}

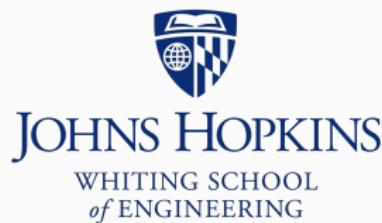
$ g++ -c circle_main.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -c circle.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o circle_main circle_main.o circle.o
$ ./circle_main
PI is 3.14
```

Design principles

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Design principles

Learning to program and learning to be a software *designer & engineer* are different things

Here we focus on programming, but some C/C++ concepts are best understood in light of design & engineering

Let's contrast programming & software engineering:

Design principles

Programming project (e.g. for this class)

- Start with: formal algorithm, detailed requirements
- Work: by yourself or on small team
- Phases: just do it
- Expected software lifetime: often brief

Software engineering

- Start with: human input, vague requirements
- Work: on large team
- Phases: user interviews, design documents, prototype, product
- Expected software lifetime: years, decades
 - E.g. Y2K bug; that software was decades old

Design principles



https://en.wikipedia.org/wiki/Year_2000_problem

An electronic sign displaying the year incorrectly as 1900 on 3 January 2000; example of Y2K bug

Design principles

Readability: easy to read and understand code

Conciseness: no needlessly complex code, little repetition

Robustness: modifying one aspect shouldn't break another

Design principles

To achieve these goals:

- *Separation of concerns*: distinct code units address distinct problems
- *Encapsulation*: implementation details are hidden; only a simple interface is exposed

Separation of concerns

Distinct code units address distinct problems

- Code units: source files, functions, classes, etc
- Also called *modularity*

Separation of concerns

Advantages:

- Promotes re-use
- Easier to develop separate code units separately
- Robust

Caveats:

- Concerns are often not perfectly separable
 - Chess: *some* part of the code needs to understand both Board and Piece to check if a move is legal
- Combining concerns can make code more efficient
 - Chess: say we want to determine whether a king is “in check” without iterating over all the pieces. Hard to do without substantially mixing Piece and Board concerns.

Encapsulation

Modular programming is an exercise in creating independent code units with good “interfaces”:

- classes have public members and protected/private members
- Functions have parameters, but they also have local variables

Part of the code unit is accessible (public members, arguments), part is hidden (protected/private, local variables)

Encapsulation

I'm writing a class, but I don't like to spend time debugging and dealing with compiler errors. Why not make everything public?

```
class GradeList {  
public:  
    void add(double grade);  
    double percentile(double percentile);  
    double mean();  
    double median();  
    std::vector<double> grades; // was private, NOW PUBLIC  
    bool is_sorted;           // was private, NOW PUBLIC  
};
```

Encapsulation

A prime duty of GradeList is to control when and how grades is sorted:

- If grades is sorted, it's easy to answer percentile queries
- If it's already sorted, it's wasteful to try to sort it again

Once grades & is_sorted are public, user can modify them and there's no guaranteed relationship between is_sorted & grades

Encapsulation

Also, say we decide to switch from `vector<double>` to `multiset<double>`

- `multiset` can always be iterated over in sorted order
- ... so we can also get rid of `is_sorted`

Now user code that depended on `grades` being a `vector<double>` (or that depended on `is_sorted` existing) won't compile

Encapsulation

An interface should be “as simple as possible, but no simpler”

- Have a few public members that handle everything users need
- Keep all details private
 - These can change later without breaking user code
 - class can maintain *invariants* over its private members
(`is_sorted` is true if and only if `grades` is sorted) without fear that user will break them

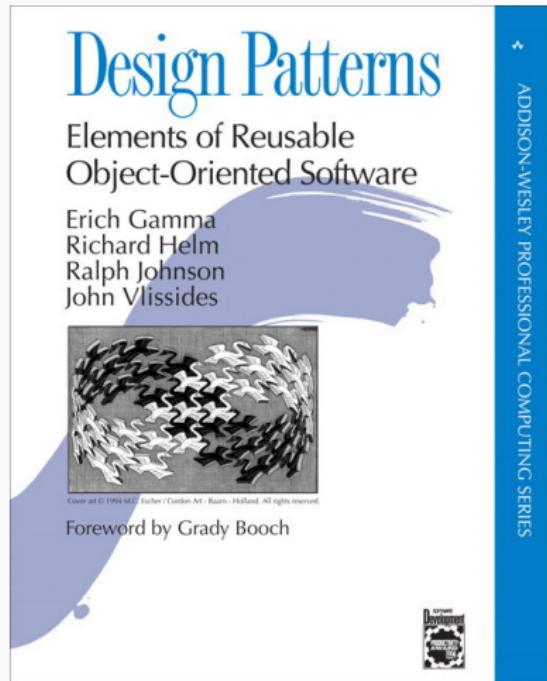
Encapsulation

Scott Meyers, a great C++ programmer & communicator, expresses prime design principle as: “Make interfaces easy to use correctly and hard to use incorrectly”

- bit.ly/meyers_iface

C++ design

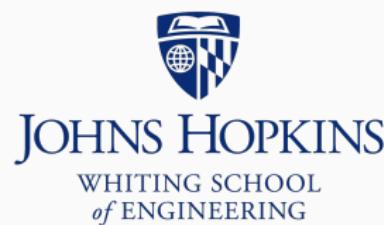
For specific C++ design strategies:



Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

stringstream

`std::stringstream` and its specialized cousins

`std::istringstream` and `std::ostringstream` help you get data
into and out of strings

First, let's see an example of overloading the extraction (`>>`)
operator

stringstream

```
std::istream& operator>>(std::istream& is, Complex& c) {  
    // Assume format "3.0 + 4.0 i"  
    string tmp;  
    is >> c.real;          // parse real coefficient  
    is >> tmp;             // skip the +  
    is >> c.imaginary;    // parse imaginary coefficient  
    is >> tmp;             // parse the i  
    assert(tmp == "i");   // sanity check  
    return is;  
}
```

Similar to operator<< but with istream instead of ostream and
>> instead of <<

Second argument must be non-const reference so we can modify

stringstream

`std::stringstream` is a stream, like `std::cout` or `std::cin`

Instead of reading or writing to console, it reads and writes to a temporary string (“buffer”) stored inside

The string buffer can be accessed with `.str()`

```
#include <string>
#include <iostream>
#include <sstream> // for std::stringstream

using std::cout;    using std::endl;
using std::string;  using std::stringstream;

int main() {
    stringstream ss;           // buffer is empty
    ss << "Hello, world!" << endl; // write message to buffer
    cout << ss.str();         // retrieve buffer with .str()
    return 0;
}
```

stringstream

```
$ g++ -c ss1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o ss1 ss1.o  
$ ./ss1  
Hello, world!
```

Why not use `std::string` and `+` operator instead?

`+` operator overload for `std::string` only handles `std::string` or `char` arguments; often we want other types too

`stringstream` works with `operator<<` and `operator>>`; an operator overload for either will work with `stringstream` just as well as with `cin/cout`

stringstream

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

using std::cout;    using std::endl;
using std::string;  using std:: stringstream;

int main() {
    stringstream ss;
    ss << "Hello" << ' ' << 35 << " world"; // mix string, char, int

    string word1, word2;
    int num;
    ss >> word1 >> num >> word2;           // read them back out
    cout << word1 << ", " << word2 << '!';
    return 0;
}
```

stringstream

```
$ g++ -c ss2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o ss2 ss2.o  
$ ./ss2  
Hello, world!
```

stringstream

```
#include <string>
#include <iostream>
#include <sstream>
#include <vector>

using std::ostream; using std::istream;
using std::cout;    using std::endl;
using std::vector;  using std::stringstream;

ostream& operator<<(ostream& os, const vector<int>& vec) {
    for(int i : vec) { os << i << ' ';}
    return os;
}

istream& operator>>(istream& is, vector<int>& vec) {
    int i;
    while(is >> i) {
        vec.push_back(i);
    }
    return is;
}
```

stringstream

```
int main() {
    stringstream ss("1 2 3 4 5");
    vector<int> vec;
    ss >> vec;
    cout << vec << endl;
    return 0;
}
```

```
$ g++ -c ss2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o ss2 ss2.o
$ ./ss2
1 2 3 4 5
```

stringstream

Whoa! What was that `for(int i : vec)` business?

```
for(int i : vec) { os << i << ' '; }
```

That's a “ranged for”, a convenience added in C++11 that we will discuss more later

stringstream

stringstream is an example of *multiple inheritance*

Inherits from:

- `istringstream`, which handles input and overloads extraction operator
- `ostringstream`, which handles output and overloads insertion operator

If you only need one or the other, you can use `istringstream` or `ostringstream`

stringstream

stringstream can be especially useful for testing, especially if you have designed your code to handle input and output as streams

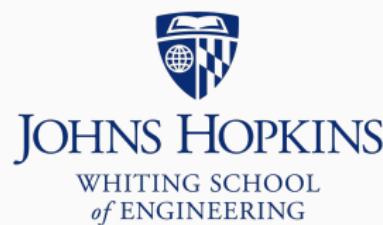
Example like today's exercise:

```
// Only using for input
istringstream ss("Eight of Hearts "
    "Ten of Hearts "
    "Jack of Hearts "
    "Nine of Hearts "
    "Queen of Hearts");
assert(straight_flush(ss));
```

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

fstream

In C, printf wrote stdout and scanf read from stdin

fprintf and fscanf were their counterparts for working with named files

In C++, we have std::cout and std::cin

std::ofstream and std::ifstream are their counterparts for working with named files

- `#include <fstream>`

ofstream / ifstream

```
#include <iostream>
#include <fstream>

using std::ofstream;
using std::endl;

int main() {
    // open for writing, create if needed, possibly overwrite
    // just like fopen("hello.txt", "w")
    ofstream ofile("hello.txt");
    if(ofile) {
        ofile << "Hello, World!" << endl;
    } else {
        std::cout << "Error opening file" << endl;
        return 1;
    }
    return 0;
}
```

ofstream / ifstream

```
$ g++ -o ofs1 ofs1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ ./ofs1  
$ cat hello.txt  
Hello, World!
```

ofstream / ifstream

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

using std::cout;
using std::endl;
using std::string;
using std::ifstream;

int main() {
    ifstream ifile("hello.txt"); // assuming we've written it
    if(ifile) {
        string word;
        while(ifile >> word) { cout << word << ' '; }
        cout << endl;
    } else {
        std::cout << "Error opening file" << endl;
        return 1;
    }
    return 0;
}
```

ofstream / ifstream

```
$ g++ -o ifs1 ifs1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ ./ifs1  
Hello, World!
```

`ofstream / ifstream`

Could have explicitly closed the file:

```
if(ifile) {  
    string word;  
    while(ifile >> word) { cout << word << ' '; }  
    cout << endl;  
    ifile.close(); // *** explicitly close ***  
}  
}
```

Not necessary; if you don't close explicitly *the fstream destructor closes it for you*

ofstream / ifstream

Unlike *streams* (`std::ostream`, `std::istream`), you can move *forward and backward* through a file

```
#include <iostream>
#include <fstream>

using std::endl;      using std::cout;
using std::ifstream;  using std::ofstream;

int main() {
    char *buffer = NULL;
    int length = 0;
    {
        ifstream is("hello.txt");
        if(!is) {
            cout << "Error opening file" << endl;
            return 1;
        }
        is.seekg(0, is.end); // move to *end* of file
        length = is.tellg(); // get my offset into file
        is.seekg(0, is.beg); // move to *beginning* of file

        buffer = new char[length];
        is.read(buffer, length); // read file contents
    } // is goes out of scope and is closed
```

ofstream / ifstream

```
// main() continued

{
    ofstream os("hello2.txt");
    if(!os) {
        cout << "Error opening file" << endl;
        delete[] buffer;
        return 1;
    }
    os.write(buffer, length);
    delete[] buffer;
} // os goes out of scope and is closed

return 0;
}
```

ofstream / ifstream

```
$ g++ -o seek seek.cpp -std=c++11 -pedantic -Wall -Wextra
$ cat hello.txt
Hello, World!
$ ./seek
$ cat hello2.txt
Hello, World!
```

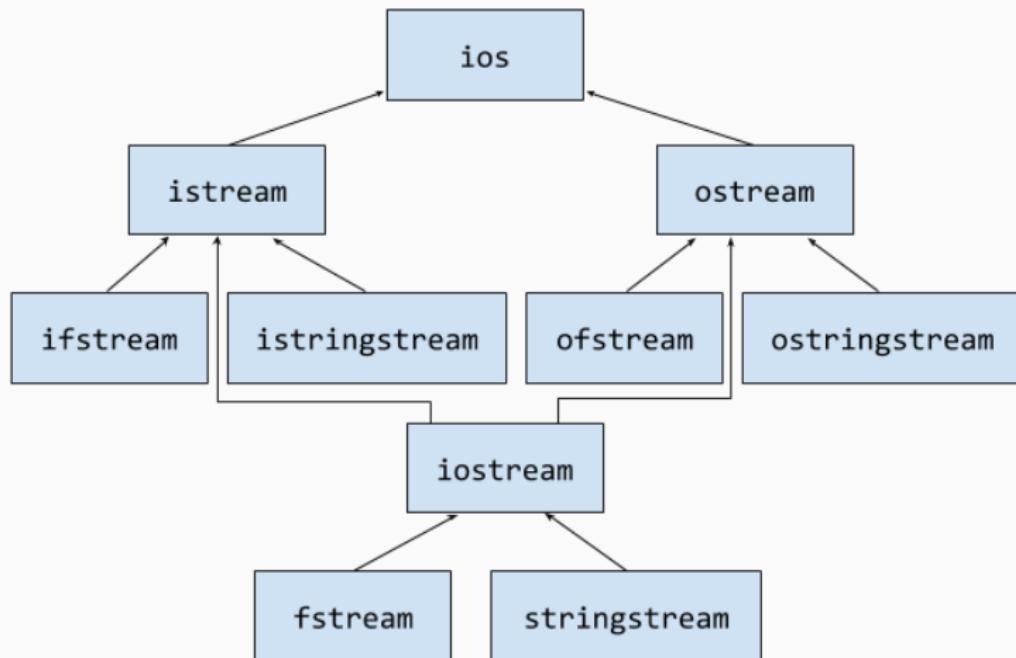
ofstream / ifstream

To summarize:

```
std::ifstream is("filename"); // open file for reading
if(is) { // check that open was successful
    is >> xyz; // read from file
    is.seekg(0, is.end); // move to *end* of file
    length = is.tellg(); // get my offset into file
    is.seekg(0, is.beg); // move to *beginning* of file
    is.read(buffer, length); // read from file (binary, like fread)
    is.close(); // close (alternately, let destructor do it)
}

std::ofstream os("filename2"); // open file for writing
if(os) { // check that open was successful
    os << xyz; // write to file (text)
    os.write(buffer, length); // write to file (binary, like fwrite)
} // destructor closes file
```

ofstream / ifstream

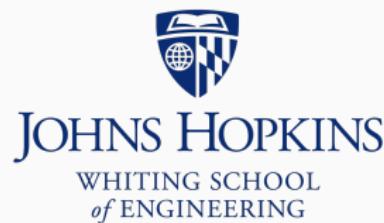


Exceptions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Exceptions

We use exceptions to indicate a *fatal* error has occurred, where there is *no reasonable way to continue from the point of the error* (the *throw point*)

It might be possible to continue from *somewhere else*, but not from the throw point

Exceptions

Behold, a bad program:

```
#include <iostream>

using std::cout;  using std::endl;

int main() {
    size_t mem = 1;
    while(true) {
        char *lots_of_mem = new char[mem];
        delete[] lots_of_mem;
        mem *= 2;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}
```

Exceptions

```
$ g++ -c exceptions1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o exceptions1 exceptions1.o  
$ ./exceptions1  
terminate called after throwing an instance of 'std::bad_alloc'  
what(): std::bad_alloc
```

Exceptions

Keeps allocating bigger arrays until an allocation fails

The exception makes sense:

- Any pointer returned by `new[]` would be unusable; program doesn't necessarily expect that
- Program can signal that it *does* expect that by *catching* the appropriate exception
 - Since we *don't* do so here, the exception crashes the program

Exceptions

```
char *lots_of_mem = new char[mem];
```

Why not have `new[]` return `NULL` on failure, like `malloc`?

- When call stack is deep: `f1() -> f2() -> f3() -> ...`
propagating errors backward requires much coordination
- If any function fails to propagate error back, chain is broken
- Error encoding must be managed (e.g. 1 = success, 2 = out of
memory, ...); no standard

Exceptions are more concise, more flexible, less error prone than
manually propagating errors through the chain of callers

Exceptions

When an exception is thrown, a `std::exception` object is created

Exception types ultimately derive from `std::exception` base class

Exception's type and contents (accessed via `.what()`) describe what went wrong

Exceptions

Looking in documentation for new/new T[n], you can see the exception thrown is of type bad_alloc

function

operator new[]

<new>

C++98

C++11



```
throwing (1) void* operator new[] (std::size_t size);
nothrow (2) void* operator new[] (std::size_t size, const std::nothrow_t& nothrow_value) noexcept;
placement (3) void* operator new[] (std::size_t size, void* ptr) noexcept;
```

Allocate storage space for array

Default allocation functions (array form).

(1) throwing allocation

Allocates *size* bytes of storage, suitably aligned to represent any object of that size, and returns a non-null pointer to the first byte of this block.

On failure, it throws a **bad_alloc** exception.

The default definition allocates memory by calling `operator new::operator new (size)`.

If replaced, both `operator new` and `operator new[]` shall return pointers with identical properties.

(2) noexcept allocation

Same as above (1), except that on failure it returns a *null pointer* instead of throwing an exception.

C++98

C++11

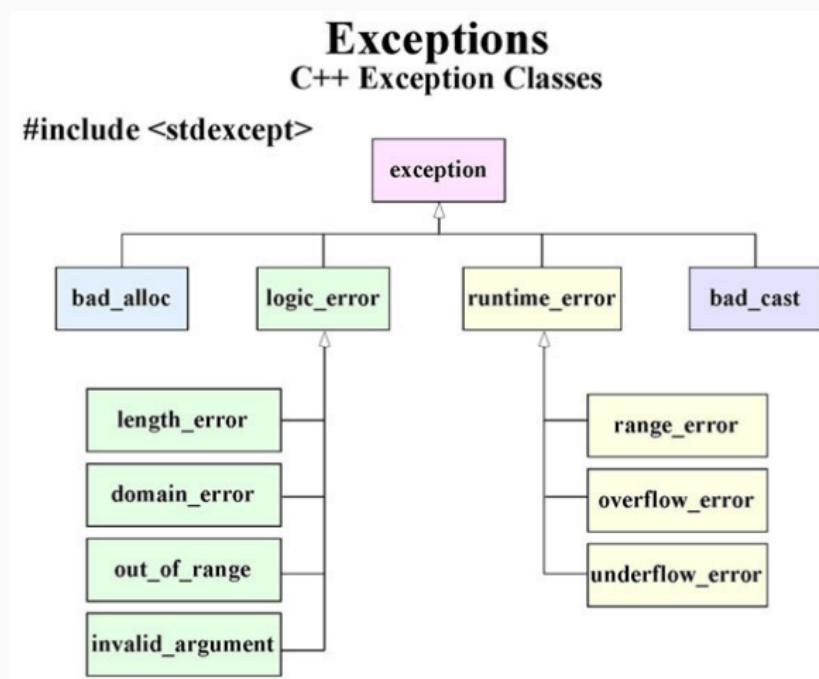


The default definition allocates memory by calling the `nothrow` version of `operator new::operator new (size,nothrow)`.

If replaced, both `operator new` and `operator new[]` shall return pointers with identical properties.

Exceptions

Standard exceptions:



Exceptions

```
#include <iostream>
#include <new> // bad_alloc defined here

using std::cout;  using std::endl;

int main() {
    size_t mem = 1;
    char *lots_of_mem;
    try {
        while(true) {
            lots_of_mem = new char[mem];
            delete[] lots_of_mem;
            mem *= 2;
        }
    }
    catch(const std::bad_alloc& ex) {
        cout << "Got a bad_alloc!" << endl
            << ex.what() << endl;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}
```

Exceptions

```
$ g++ -c exceptions2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o exceptions2 exceptions2.o  
$ ./exceptions2  
Got a bad_alloc!  
std::bad_alloc  
Forever is a long time
```

Exceptions

Another example:

```
#include <iostream>
#include <vector>
#include <stdexcept> // standard exception classes defined

using std::cout;    using std::endl;
using std::vector;

int main() {
    vector<int> vec = {1, 2, 3};
    try {
        cout << vec.at(3) << endl;
    } catch(const std::out_of_range& e) {
        cout << "Exception: " << endl << e.what() << endl;
    }
    return 0;
}
```

Exceptions

```
$ g++ -c exceptions3.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o exceptions3 exceptions3.o  
$ ./exceptions3  
Exception:  
vector::_M_range_check: __n (which is 3) >= this->size() (which is 3)
```

Exceptions

try marks block of code where an exception might be thrown

```
try {  
    while(true) {  
        lots_of_mem = new char[mem]; // !  
        delete[] lots_of_mem;  
        mem *= 2;  
    }  
}
```

Tells C++ “exceptions might be thrown, and I’m ready to handle some or all of them”

Exceptions

catch block, immediately after try block, says what to do in the event of a particular exception

```
catch(const bad_alloc& ex) {  
    cout << "Yep, got a bad_alloc" << endl;  
}
```

Doesn't strictly have to be a const reference; could be normal reference. But we rarely modify the exception object.

Exceptions

When exception is thrown, we don't proceed to the next statement
Instead we follow a process of “unwinding”

Exceptions

Unwinding: keep moving “up” to wider enclosing scopes; stop at try block with relevant catch clause

```
if(a == b) {
    try {
        while(c < 10) {
            try {
                if(d % 3 == 1) {
                    throw std::runtime_error("!");
                }
            }
            catch(const bad_alloc &e) {
                ...
            }
        }
    }
    catch(const runtime_error &e) {
        // after throw, control moves here
        ...
    }
}
```

Exceptions

If we unwind all the way to the point where our scope is an entire function, we jump back to the caller and continue the unwinding

Exceptions

```
void fun2() { // (called by fun1)
    while(...) {
        try {
            // unwinding from here...
            throw std::runtime_error("whoa");
        } catch(const bad_alloc& e) {
            // only catches bad_alloc, not runtime_error
            ...
        }
    }
}

void fun1() {
    try {
        fun2();
    } catch(const runtime_error& e) {
        // ends up here...
        ...
    }
}
```

Exceptions

If exception is never caught – i.e. we unwind all the way through main – exception info is printed to console & program exits

That's what happened in the original bad_alloc example

Exceptions

```
#include <iostream>

using std::cout;  using std::endl;

int main() {
    size_t mem = 1;
    while(true) {
        char *lots_of_mem = new char[mem];
        delete[] lots_of_mem;
        mem *= 2;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}

$ g++ -o exceptions1 exceptions1.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./exceptions1
terminate called after throwing an instance of 'std::bad_alloc'
what():  std::bad_alloc
```

Exceptions

```
#include <iostream>
#include <stdexcept>

using std::cout;  using std::endl;

void fun2() {
    cout << "fun2: top" << endl;
    throw std::runtime_error("runtime_error in fun2");
    cout << "fun2: bottom" << endl;
}

void fun1() {
    cout << "fun1: top" << endl;
    fun2();
    cout << "fun1: bottom" << endl;
}

int main() {
    try {
        cout << "main: try top" << endl;
        fun1();
        cout << "main: try bottom" << endl;
    } catch(const std::runtime_error &error) {
        cout << "Exception handled in main: " << error.what() << endl;
    }
    cout << "main: bottom" << endl;
    return 0;
}
```

Exceptions

```
$ g++ -c except_unwind.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o except_unwind except_unwind.o
$ ./except_unwind
main: try top
fun1: top
fun2: top
Exception handled in main: runtime_error in fun2
main: bottom
```

Exceptions

Unwinding causes local variables to go out of scope

Destructor is called when object goes out of scope, regardless of whether exit is due to return, break, continue, exception, ...

Exceptions

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

// Prints messages upon construction and destruction
// For investigating when constructors/destructors are called
class HelloGoodbye {
public:
    HelloGoodbye(const std::string& nm) : name(nm) {
        std::cout << name << ": hello" << std::endl;
    }

    ~HelloGoodbye() {
        std::cout << name << ": goodbye" << std::endl;
    }

private:
    std::string name;
};

};
```

Exceptions

```
#include <iostream>
#include <stdexcept>
#include "hello_goodbye.h"

using std::cout;  using std::endl;

void fun2() {
    HelloGoodbye fun2_top("fun2_top");
    throw std::runtime_error("runtime_error in fun2");
    HelloGoodbye fun2_bottom("fun2_bottom");
}

void fun1() {
    HelloGoodbye fun1_top("fun1_top");
    fun2();
    HelloGoodbye fun1_bottom("fun1_bottom");
}

int main() {
    try {
        HelloGoodbye main_top("main_top");
        fun1();
        HelloGoodbye main_bottom("main_bottom");
    }
    catch(const std::runtime_error &error) {
        cout << "Exception handled in main: " << error.what() << endl;
    }
    return 0;
}
```

Exceptions

```
$ g++ -c except_unwind2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o except_unwind2 except_unwind2.o
$ ./except_unwind2
main_top: hello
fun1_top: hello
fun2_top: hello
fun2_top: goodbye
fun1_top: goodbye
main_top: goodbye
Exception handled in main: runtime_error in fun2
```

Exceptions

C++ passes control to *first* catch block whose type equals or is a base class of the thrown exception

Arrange catch blocks from *most to least specific* type

- E.g. `catch(const std::runtime_error& e)` before
`catch(const std::exception& e)`

Exceptions

```
#include <iostream>
#include <stdexcept>

using std::cout;  using std::endl;

int main() {
    try {
        throw std::out_of_range("not a runtime_error");
        cout << "no exception" << endl;
    } catch(const std::runtime_error& e) {
        cout << "runtime_error: " << e.what() << endl;
    } catch(const std::exception& e) {
        // out_of_range is derived from exception
        // but *not* from runtime_error
        cout << "exception: " << e.what() << endl;
    }
    return 0;
}
```

Exceptions

```
$ g++ -c exc_spec.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o exc_spec exc_spec.o  
$ ./exc_spec  
exception: not a runtime_error
```

Less specific catch(const std::exception& e) block is used

Exceptions

You can define your own exception class, derived from exception

Since exceptions are related through inheritance, you can choose whether to catch a base class (thereby catching more different things) or a derived class

Following card-game example demonstrates both points

Exceptions: card_game.h part 1

```
#ifndef CARD_GAME_H
#define CARD_GAME_H

#include <iostream>
#include <sstream>
#include <stdexcept>
#include <vector>
#include <utility>
#include <string>
#include <algorithm>

enum class Suit { HEART, DIAMOND, SPADE, CLUB };

enum class Rank { ACE = 1, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
                 EIGHT, NINE, TEN, JACK, QUEEN, KING };

// Card = suit + rank
typedef std::pair<Suit, Rank> Card;
```

Exceptions: card_game.h part 2

```
class BadCardError : public std::runtime_error {
public:
    BadCardError(Card c) :
        std::runtime_error("bad card"), card(c) { }
private:
    Card card;
};
```

Exceptions: card_game.h part 3

```
class CardGame {
public:
    CardGame() : deck(), discard_pile() {
        for(int s = (int)Suit::HEART; s <= (int)Suit::CLUB; s++) {
            for(int r = (int)Rank::ACE; r <= (int)Rank::KING; r++) {
                deck.push_back(std::make_pair((Suit)s, (Rank)r));
            }
        }
        std::random_shuffle(deck.begin(), deck.end());
    }

    Card draw();           // user takes card from deck
    void discard(Card c); // user puts card on discard pile

    size_t deck_size() const { return deck.size(); }

private:
    std::vector<Card> deck, discard_pile;
};

#endif // CARD_GAME_H
```

Exceptions: card_game.cpp

```
#include "card_game.h"

Card CardGame::draw() {
    Card c = deck.back();
    deck.pop_back();
    return c;
}

void CardGame::discard(Card c) {
    // sanity check the card first
    if(c.first < Suit::HEART || c.first > Suit::CLUB ||
       c.second < Rank::ACE || c.second > Rank::KING)
    {
        throw BadCardError(c);
    }
    discard_pile.push_back(c);
}
```

Exceptions: card_game_main1.cpp

```
#include "card_game.h"

using std::cout;  using std::endl;

int main() {
    CardGame cg;
    Card c = cg.draw();
    try {
        cg.discard(c);
        cout << "no exception" << endl;
    } catch(const std::runtime_error& e) {
        cout << "runtime_error: " << e.what() << endl;
    }
    return 0;
}
```

Exceptions

```
$ g++ -o card_game_main1 card_game_main1.cpp card_game.cpp  
$ ./card_game_main1  
no exception
```

Exceptions: card_game_main2.cpp

```
#include "card_game.h"

using std::cout;  using std::endl;

int main() {
    CardGame cg;
    Card c = cg.draw();
    try {
        c.first = (Suit)5; // Card is now malformed!
        cg.discard(c);
        cout << "no exception" << endl;
    } catch(const std::runtime_error& e) {
        cout << "runtime_error: " << e.what() << endl;
    }
    return 0;
}
```

Exceptions

```
$ g++ -o card_game_main2 card_game_main2.cpp card_game.cpp  
$ ./card_game_main2  
runtime_error: bad card
```

Our catch block caught the exception

Exceptions: card_game_main3.cpp

```
#include "card_game.h"

using std::cout;  using std::endl;

int main() {
    CardGame cg;
    Card c = cg.draw();
    try {
        c.first = (Suit)5;
        cg.discard(c);
        cout << "no exception" << endl;
    } catch(const std::runtime_error& e) {
        // first catch block that either equals or is a
        // base class of the thrown exception is the one
        // used
        cout << "runtime_error: " << e.what() << endl;
    } catch(const std::exception& e) {
        cout << "exception: " << e.what() << endl;
    }
    return 0;
}
```

Exceptions

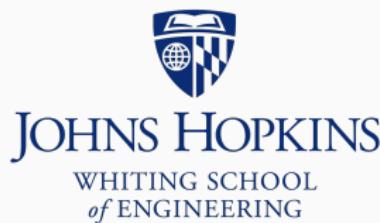
```
$ g++ -o card_game_main3 card_game_main3.cpp card_game.cpp  
$ ./card_game_main3  
runtime_error: bad card
```

Copying, assignment and the Rule of 3

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Copying, assignment and the Rule of 3

We know there is a difference between == and =

But there are two kinds of =:

- = in a declaration, like int a = 4; (initialization)
- = elsewhere, like a = 4; (assignment)

```
Complex c = {3.0, 2.0};      // = in declaration: *initialization*
Complex c2 = c;              // (same)
c = Complex(4.0, 5.0);       // = outside declaration: *assignment*
if(c2.get_real() == 3.0) {   // == is *equality testing*
    // ...
}
```

image.h

Image has resources managed by the constructor & destructor:

```
class Image {
public:
    Image(const char *orig, int r, int c) : nrow(r), ncol(c) {
        image = new char[r*c];
        for(int i = 0; i < nrow * ncol; i++) {
            image[i] = orig[i];
        }
    }

    ~Image() { delete[] image; }

    const char *get_image() const { return image; }
    int get_nrow() const { return nrow; }
    int get_ncol() const { return ncol; }

    void set_pixel(char pix, int row, int col) {
        image[row * ncol + col] = pix;
    }
private:
    char *image; // image data
    int nrow, ncol; // # rows and columns
};

std::ostream& operator<<(std::ostream&, const Image&);
```

image.cpp

```
#include <iostream>
#include "image.h"

using std::endl;
using std::ostream;

ostream& operator<<(ostream& os, const Image& image) {
    for(int i = 0; i < image.get_nrow(); i++) {
        for(int j = 0; j < image.get_ncol(); j++) {
            os << image.get_image()[i*image.get_ncol()+j] << ' ';
        }
        os << endl;
    }
    return os;
}
```

image_main.cpp

```
#include <iostream>
#include "image.h"
using std::cout;  using std::endl;

int main() {
    Image x_wins("X-O-XO--X", 3, 3);
    cout << x_wins << "** X wins! **" << endl;
    return 0;
}
```

```
$ g++ -o image_main image_main.cpp image.cpp
$ ./image_main
X - O
- X O
- - X
** X wins! **
```

image_main2.cpp

```
#include <iostream>
#include "image.h"

using std::cout;  using std::endl;

int main() {
    Image x_wins("X-O-XO--X", 3, 3);
    Image o_wins = x_wins;
    o_wins.set_pixel('0', 2, 2); // set bottom right to '0'
    cout << x_wins << "** X wins! **" << endl << endl;
    cout << o_wins << "** O wins! **" << endl;
    return 0;
}
```

image_main2.cpp

```
$ g++ -o image_main2 image_main2.cpp image.cpp  
$ ./image_main2  
X - 0  
- X 0  
- - 0  
** X wins! **
```

```
X - 0  
- X 0  
- - 0  
** O wins! **
```

Oops, both have 0 in bottom right corner

`o_wins.set_pixel(...)` affected both `x_wins` & `o_wins`!

image_main2.cpp

Also: destructor delete[]s the same pointer twice

```
$ valgrind ./image_main2 > /dev/null
==42== Memcheck, a memory error detector
==42== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==42== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==42== Command: ./image_main2
==42==
==42== Jump to the invalid address stated on the next line
==42==    at 0x0: ???
==42==    by 0x4008C9: _start (in /app/750_ruleof3/image_main2)
==42==    by 0x1FFF000C37: ???
==42==    by 0x4228F7F: ??? (in /usr/lib64/ld-2.26.so)
==42== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==42==
==42==
==42== Process terminating with default action of signal 11 (SIGSEGV)
==42== Bad permissions for mapped region at address 0x0
==42==    at 0x0: ???
==42==    by 0x4008C9: _start (in /app/750_ruleof3/image_main2)
==42==    by 0x1FFF000C37: ???
==42==    by 0x4228F7F: ??? (in /usr/lib64/ld-2.26.so)
==42==
==42== HEAP SUMMARY:
==42==     in use at exit: 0 bytes in 0 blocks
==42== total heap usage: 1 allocs, 1 frees, 72,704 bytes allocated
==42==
==42== All heap blocks were freed -- no leaks are possible
```

Initialization & assignment

Image o_wins = x_wins; does *shallow copy*

- Copies x_wins.image pointer directly into o_wins.image, so both are using same heap array
- Instead, we want *deep copy*; o_wins should be a new buffer, with contents of x_wins copied over
- Want this both for initialization and for assignment

Image x_wins("X-O-XO--X", 3, 3);

Image o_wins = x_wins;

Rule of 3

Image is an example of a class that manages resources, and therefore has a *non-trivial destructor*

Rule of 3: If you have to manage how an object is destroyed, you should also manage how it's copied

Rule of 3 (technical version): If you have a non-trivial destructor, you should also define a *copy constructor* and operator=

Case in point: Image should be deep copied

Rule of 3

Copy constructor initializes a class variable as a copy of another
operator= is called when one object is assigned to another

```
Complex c = {3.0, 2.0}; // non-default constructor
Complex c2 = c;          // copy constructor
c = Complex(4.0, 5.0);   // non-default ctor for right-hand side
                        // operator= to copy into left-hand side
```

Copy constructor

Copy constructor is called when:

- Initializing:
 - `Image o_wins = x_wins;`
 - `Image o_wins(x_wins);` (same meaning as above)
- Passing by value
- Returning by value

Copy constructor

Copy constructor for Image:

```
Image(const Image& o) : nrow(o.nrow), ncol(o.ncol) {  
    // Do a *deep copy*, similarly to the  
    // non-default constructor  
    image = new char[nrow * ncol];  
    for(int i = 0; i < nrow * ncol; i++) {  
        image[i] = o.image[i];  
    }  
}
```

operator=

operator= is called when assigning one class variable to another

- Except for initialization; copy constructor handles that

```
Image& operator=(const Image& o) {
    delete[] image; // deallocate previous image memory
    nrow = o.nrow;
    ncol = o.ncol;
    image = new char[nrow * ncol];
    for(int i = 0; i < nrow * ncol; i++) {
        image[i] = o.image[i];
    }
    return *this; // for chaining
}
```

It's a normal member function, not a constructor, so we can't use initializer list syntax

Rule of 3

If you don't specify copy constructor or operator=, compiler adds *implicit* version that *shallow copies*

- Simply copies each field
- class field will have its copy constructors or operator= function called
- Pointer to heap memory will simply be copied, without the heap memory itself being copied

Another way of stating the Rule of 3: if your class has a non-trivial destructor, you probably *don't* want shallow copying

Rule of 3

When we add the copy constructor and operator= defined above, we get the expected behavior:

```
$ g++ -o image_fixed image_fixed.cpp image.cpp
```

```
$ ./image_fixed
```

```
X - O
```

```
- X O
```

```
- - X
```

```
** X wins! **
```

```
X - O
```

```
- X O
```

```
- - O
```

```
** O wins! **
```

Rule of 3

And no complaints from valgrind:

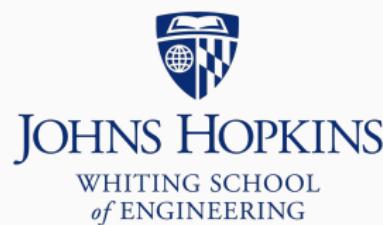
```
$ valgrind ./image_fixed > /dev/null
==52== Memcheck, a memory error detector
==52== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==52== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==52== Command: ./image_fixed
==52==
==52==
==52== HEAP SUMMARY:
==52==     in use at exit: 0 bytes in 0 blocks
==52==   total heap usage: 4 allocs, 4 frees, 76,818 bytes allocated
==52==
==52== All heap blocks were freed -- no leaks are possible
==52==
==52== For counts of detected and suppressed errors, rerun with: -v
==52== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Template functions

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Template functions

Templates allow us to write function or class once:

```
template<typename T>
void print_array(const T* a, int count) { ... }
```

but get a whole *family* of *overloaded specializations*:

```
void print_array(const int* a, int count) { ... }
void print_array(const float* a, int count) { ... }
void print_array(const char* a, int count) { ... }
void print_array(const linked_list_node* a, int count) { ... }
...
...
```

Template functions

This function sums even-indexed elements in a list:

```
int sum_every_other(const vector<int>& ls) {
    int total = 0;
    for(vector<int>::const_iterator it = ls.cbegin();
        it != ls.cend(); ++it)
    {
        total += *it;
        ++it;
    }
    return total;
}
```

Works for `const vector<int>&`, but similar code could be used for other containers

Template functions

```
#include <iostream>
#include <vector>
#include <list>

int sum_every_other_vector(const std::vector<int>& ls) {
    //
    int total = 0;
    for(std::vector<int>::const_iterator it = ls.cbegin(); it != ls.cend(); ++it) {
        //
        total += *it;
        ++it;
    }
    return total;
}

int sum_every_other_list(const std::list<int>& ls) {
    //
    int total = 0;
    for(std::list<int>::const_iterator it = ls.cbegin(); it != ls.cend(); ++it) {
        //
        total += *it;
        ++it;
    }
    return total;
}
```

Template functions

```
int main() {
    std::vector<int> vec = {10, 7, 10, 7, 10, 7};
    int sum = sum_every_other_vector(vec);
    cout << "sum of every-other (vector): " << sum << endl;

    std::list<int> lis;
    lis.assign(vec.begin(), vec.end());
    sum = sum_every_other_list(lis);
    cout << "sum of every-other (list): " << sum << endl;
    return 0;
}

$ g++ -c seo_vec_list_1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o seo_vec_list_1 seo_vec_list_1.o
$ ./seo_vec_list_1
sum of every-other (vector): 30
sum of every-other (list): 30
```

Template functions

Repetitive code is a sign of bad design

E.g. a correction for the `_vector` version also has to be made for the `_list` version (and any others we've made)

In fact, can you spot the error in our
`sum_every_other_vector`/`sum_every_other_list`?

Template functions

Extra `++it` skips over `ls.cend()` when the container has odd # elements. Need another check:

```
int sum_every_other_vector(const vector<int>& ls) {
    int total = 0;
    for(vector<int>::const_iterator it = ls.cbegin();
        it != ls.cend(); ++it)
    {
        total += *it;
        // now we can't skip over ls.cend()
        if(++it == ls.cend()) { break; } // that's better
    }
    return total;
}
```

Template functions

Template function:

```
template<typename T>
int sum_every_other(const T& ls) {
    int total = 0;
    for(typename T::const_iterator it = ls.cbegin();
        it != ls.cend(); ++it)
    {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

If we pass `vector<int>`, compiler *instantiates* an appropriate function overload

- Same if we pass `list<int>`, `vector<double>`, ...

Template functions

```
int main() {
    vector<int> vec = {10, 7, 10, 7, 10, 7};

    // ** calls template function with T=vector<int> **
    int sum = sum_every_other(vec);

    cout << "sum of every-other (vector): " << sum << endl;

    list<int> lis;
    lis.assign(vec.begin(), vec.end());

    // ** calls template function with T=list<int> **
    sum = sum_every_other(lis);

    cout << "sum of every-other (list): " << sum << endl;
    return 0;
}
```

Template functions

```
$ g++ -c seo_vec_list_2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o seo_vec_list_2 seo_vec_list_2.o  
$ ./seo_vec_list_2  
sum of every-other (vector): 30  
sum of every-other (list): 30
```

Template functions

The reason for typename here is subtle

```
for(typename T::const_iterator it = ls.cbegin();  
// ^^^^^^  
    it != ls.cend(); ++it)
```

Without typename, compiler can't distinguish whether
T::const_iterator is a type or a static field

Template functions

Without typename before T::const_iterator we get an error:

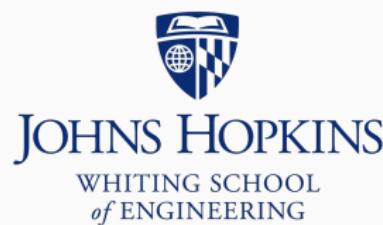
```
$ g++ -c seo_vec_list_3.cpp -std=c++11 -pedantic -Wall -Wextra
seo_vec_list_3.cpp: In function 'int sum_every_other(const T&)':
seo_vec_list_3.cpp:11:9: error: need 'typename' before 'T:: const_iterator'
because 'T' is a dependent scope
    for(T::const_iterator it = ls.cbegin());
               ^
seo_vec_list_3.cpp:11:27: error: expected ';' before 'it'
    for(T::const_iterator it = ls.cbegin();
               ^
seo_vec_list_3.cpp:12:9: error: 'it' was not declared in this scope
    it != ls.cend(); ++it)
               ^
seo_vec_list_3.cpp:12:9: note: suggested alternative: 'int'
    it != ls.cend(); ++it)
               ^
int
seo_vec_list_3.cpp: In instantiation of 'int sum_every_other(const T&) [with T =
std::vector<int>]':
seo_vec_list_3.cpp:25:34:   required from here
seo_vec_list_3.cpp:11:43: error: dependent-name 'T:: const_iterator' is parsed
as a non-type, but instantiation yields a type
    for(T::const_iterator it = ls.cbegin());
               ^
seo_vec_list_3.cpp:11:43: note: say 'typename T:: const_iterator' if a type is
meant
seo_vec_list_3.cpp: In instantiation of 'int sum_every_other(const T&) [with T =
```

Template classes

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Template classes

When we saw STL, we considered these templates:

```
template<typename T>
struct Node {
    T payload;
    Node *next;
};

template<typename T>
void print_list(Node<T> *head) {
    Node<T> *cur = head;
    while(cur != NULL) {
        cout << cur->payload << " ";
        cur = cur->next;
    }
    cout << endl;
}
```

One struct/function that works for (almost) any type T

Template classes

```
int main() {
    Node<float> f3 = {95.1f, NULL}; // float payload
    Node<float> f2 = {48.7f, &f3}; // float payload
    Node<float> f1 = {24.3f, &f2}; // float payload
    print_list(&f1);

    Node<int> i2 = {239, NULL}; // int payload
    Node<int> i1 = {114, &i2}; // int payload
    print_list(&i1);

    return 0;
}
```

Template classes

```
$ g++ -c ll_template_cpp.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o ll_template_cpp ll_template_cpp.o  
$ ./ll_template_cpp  
24.3 48.7 95.1  
114 239
```

Now we make it a template class.

Template classes

```
#include <iostream>

template<typename T>
class Node {
public:
    Node(T pay, Node<T> *nx) : payload(pay), next(nx) { }

    void print() const {
        const Node<T> *cur = this;
        while(cur != NULL) {
            std::cout << cur->payload << ' ';
            cur = cur->next;
        }
        std::cout << std::endl;
    }

private:
    T payload;
    Node<T> *next;
};
```

Template classes

```
#include "ll_temp.h"

int main() {
    Node<float> f3 = {95.1f, NULL};
    Node<float> f2 = {48.7f, &f3};
    Node<float> f1 = {24.3f, &f2};
    f1.print();

    Node<int> i2 = {239, NULL};
    Node<int> i1 = {114, &i2};
    i1.print();

    return 0;
}

$ g++ -o ll_temp_main ll_temp_main.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./ll_temp_main
24.3 48.7 95.1
114 239
```

Template instantiation

Compiler acts *lazily* when instantiating templates

Doesn't instantiate until *first use*

```
Node<float> f3 = {95.1f, NULL}; // OK fine, I'll instantiate Node<float>
Node<float> f2 = {48.7f, &f3};
Node<float> f1 = {24.3f, &f2};
f1.print();                  // OK fine, I'll instantiate Node<float>::print

Node<int> i2 = {239, NULL};      // OK fine, I'll instantiate Node<int>
Node<int> i1 = {114, &i2};
i1.print();                  // OK fine, I'll instantiate Node<int>::print
```

Template instantiation

When instantiating, compiler needs the relevant template classes and functions *to be fully defined already*...

... *in contrast to typical function or class use*, where definition can be in separate .cpp files

Template instantiation

Let's move definition of print into a separate .cpp

```
// ll_temp2.h
#include <iostream>

template<typename T>
class Node {
public:
    Node(T pay, Node<T> *nx) : payload(pay), next(nx) { }
    void print() const;
private:
    T payload;
    Node<T> *next;
};
```

Template instantiation

```
// ll_temp2.cpp
#include "ll_temp2.h"

template<typename T>
void Node<T>::print() const {
    const Node<T> *cur = this;
    while(cur != NULL) {
        std::cout << cur->payload << ' ';
        cur = cur->next;
    }
    std::cout << std::endl;
}
```

Template instantiation

```
#include "ll_temp2.h"

int main() {
    Node<float> f3 = {95.1f, NULL}; // instantiate Node<float>
    Node<float> f2 = {48.7f, &f3};
    Node<float> f1 = {24.3f, &f2};
    f1.print(); // instantiate Node<float>::print *** will this work? ***

    Node<int> i2 = {239, NULL}; // instantiate Node<float>
    Node<int> i1 = {114, &i2};
    i1.print(); // instantiate Node<int>::print *** will this work? ***

    return 0;
}
```

Template instantiation

```
$ g++ ll_temp_main2.cpp ll_temp2.cpp -std=c++11 -pedantic -Wall -Wextra  
/tmp/ccEtfPAy.o: In function `main':  
ll_temp_main2.cpp:(.text+0x5f): undefined reference to `Node<float>::print()  
const'  
ll_temp_main2.cpp:(.text+0x96): undefined reference to `Node<int>::print()  
const'  
collect2: error: ld returned 1 exit status
```

Lazily instantiating `Node<float>` & `Node<int>` is fine

- Both are defined in the header

Instantiating `Node<float>::print()` & `Node<int>::print()`
won't work

- They are in a separate source file not #included here
- By convention, we never #include .cpp files

Template instantiation

Couple possible solutions:

- Move `Node<T>::print()` definition back into the header
- Put member function definitions in a separate file and
`#include` it, but *don't* give it a `.cpp` extension

Template instantiation

```
// ll_temp2.inc
template<typename T>
void Node<T>::print() const {
    const Node<T> *cur = this;
    while(cur != NULL) {
        std::cout << cur->payload << ' ';
        cur = cur->next;
    }
    std::cout << std::endl;
}
```

Template instantiation

```
#include "ll_temp2.h"    // template class definition
#include "ll_temp2.inc"  // template class member function definitions

int main() {
    Node<float> f3 = {95.1f, NULL}; // instantiate Node<float>
    Node<float> f2 = {48.7f, &f3};
    Node<float> f1 = {24.3f, &f2};
    f1.print(); // instantiate Node<float>::print *** will this work? ***

    Node<int> i2 = {239, NULL};      // instantiate Node<float>
    Node<int> i1 = {114, &i2};
    i1.print(); // instantiate Node<int>::print *** will this work? ***

    return 0;
}
```

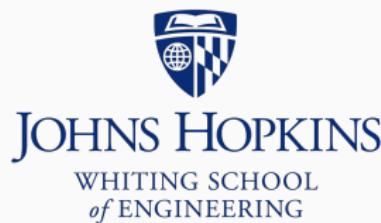
```
$ g++ -o ll_temp_main3 ll_temp_main3.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./ll_temp_main3
24.3 48.7 95.1
114 239
```

Abstract classes

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Abstract classes

What does the = 0 mean here?

```
class Piece {  
public:  
    ...  
    virtual bool legal_move_shape(CharPair, CharPair) const = 0;  
    // ^^^^  
  
    virtual bool legal_capture_shape(CharPair start,  
                                    CharPair end) const  
    {  
        return legal_move_shape(start, end);  
    }  
    ...  
};
```

Pure virtual functions

Declaring a virtual member function and adding = 0 makes it a *pure virtual function*

When we declare a pure virtual function:

- We do not give it an implementation
- It makes the class it's declared in an *abstract class*
 - We cannot create a new object with the type, though we may be able create an object with a derived *type*

Pure virtual functions

```
class Shape {  
public:  
    virtual double size() const = 0;  
    ...  
};  
  
class Shape2D : public Shape {  
    ...  
};  
  
class Circle : public Shape2D {  
public:  
    virtual double size() const {  
        return 3.14 * r * r;  
    }  
    ...  
private:  
    double r;  
    ...  
};
```

Pure virtual functions

```
int main() {
    Circle c(1.0 / sqrt(3.14));
    cout << c.size() << endl;
    return 0;
}
```

```
$ g++ -c shape_virt.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o shape_virt shape_virt.o
$ ./shape_virt
1
```

Pure virtual functions

```
int main() {
    Shape s;
    return 0;
}
```

```
$ g++ -c shape_virt2.cpp -std=c++11 -pedantic -Wall -Wextra
shape_virt2.cpp: In function 'int main()':
shape_virt2.cpp:25:11: error: cannot declare variable 's' to be of abstract type
'Shape'
    Shape s;
           ^
shape_virt2.cpp:7:7: note: because the following virtual functions are pure
within 'Shape':
class Shape {
    ~~~~
shape_virt2.cpp:9:20: note:     virtual double Shape::size() const
    virtual double size() const = 0;
        ~~~~
```

Pure virtual functions

```
int main() {
    Shape2D s2d;
    return 0;
}
```

```
$ g++ -c shape_virt3.cpp -std=c++11 -pedantic -Wall -Wextra
shape_virt3.cpp: In function 'int main()':
shape_virt3.cpp:25:13: error: cannot declare variable 's2d' to be of abstract
type 'Shape2D'
    Shape2D s2d;
           ^
shape_virt3.cpp:12:7: note: because the following virtual functions are pure
within 'Shape2D':
class Shape2D : public Shape { };
           ^
shape_virt3.cpp:9:20: note:     virtual double Shape::size() const
    virtual double size() const = 0;
           ^~~~
```

Pure virtual functions

"Cannot declare variable s to be of abstract type Shape"

When a class has one or more pure virtual functions, it cannot be instantiated; it is *abstract*

- Similar to abstract class and interface in Java

The derived Circle class *can* be instantiated because it provides an implementation for the (only) pure virtual, size()

Abstract classes

Another way to make a class abstract is give it only non-public constructors

```
class Piece {  
public:  
    ...  
protected:  
    // This is the only constructor  
    Piece( bool is_white ) : _is_white( is_white ){ }  
    ...  
};
```

Can't instantiate Piece because constructor can't be called from the outside

Abstract classes

Derived class can still use protected constructor in base class:

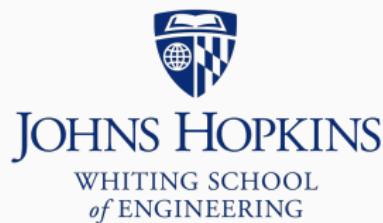
```
class Knight : public Piece {  
    ...  
public:  
    // Knight constructor calls Piece constructor  
    // OK because it's protected  
    Knight( bool is_white ) : Piece( is_white ) {}  
    ...  
};
```

Containers

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Containers

We saw STL containers and learned what they have in common:

- `container<...>::iterator` is the iterator type
 - Also `::const_iterator`, `::reverse_iterator`, ...
- `.begin()` return iterator “pointing” to beginning
- `.end()` return iterator “pointing” just past end
 - `.cbegin() / .cend()` for `const_iterator`
 - `.rbegin() / .rend()` for `reverse_iterator`

Containers

```
#include <iostream>
#include <vector>

using std::cout;  using std::endl;
using std::vector;

int main() {
    vector<int> vec = {2, 4, 6, 8};
    // Iterate forward
    for(vector<int>::iterator it = vec.begin();
        it != vec.end(); ++it)
    {
        cout << *it << ' ';
    }
    cout << endl;
    // Iterate backward
    for(vector<int>::reverse_iterator it = vec.rbegin();
        it != vec.rend(); ++it)
    {
        cout << *it << ' ';
    }
    cout << endl;
    return 0;
}
```

Containers

```
$ g++ -c iter_eg.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o iter_eg iter_eg.cpp  
$ ./iter_eg  
2 4 6 8  
8 6 4 2
```

Containers

STL containers also have:

- `container<...>::size_type`
 - Integer type for indexing container items
 - Almost always `size_t`
- `container<...>::value_type`
 - Type of items in the container
 - `std::vector<T>::value_type` is `T`
 - `std::map<K, V>::value_type` is `std::pair<K, V>`

Implementing an iterator

How would we *implement* an iterator for a container?

Begin with simple vector-like class:

Implementing an iterator

```
template <typename T>
class Vec {
public:
    // TODO: iterator typedefs: ::iterator, ::const_iterator, ...

    typedef size_t size_type;
    typedef T value_type;

    Vec() { ... }

    void push_back(const T& item) { ... }
    T pop() { ... }

    // TODO: iterator accessors: begin(), end(), cbegin(), ...

private:
    size_t size;      // # elts in vector
    size_t reserved; // # elts allocated in data array
    T *data;         // points to beginning of data array
};
```

Implementing an iterator

Can we simply use pointers? Let's try

Iterator type is T*:

```
template <typename T>
class Vec {
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    ...
};
```

Implementing an iterator

.begin() and .end() return pointers to the appropriate locations

```
template <typename T>
class Vec {
public:
    ...
    T* begin() { return data; }
    T* end() { return data + size; }

    const T* cbegin() const { return data; }
    const T* cend() const { return data + size; }

private:
    size_t size;      // # elts in vector
    size_t reserved; // # elts allocated in data array
    T *data;          // points to beginning of data array
};
```

Implementing an iterator

```
int main() {
    Vec<int> vec;
    vec.push(2); vec.push(4); vec.push(6); vec.push(8);
    // Iterate forward
    for(Vec<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        cout << *it << ' ';
    }
    cout << endl;
    return 0;
}
```

```
$ g++ -o vec_iter vec_iter.cpp -std=c++11 -pedantic -Wall -Wextra
$ ./vec_iter
2 4 6 8
```

Implementing an iterator

Simply returning a pointer works as long as elements are laid out consecutively in memory

- Fine for Vec or std::vector
- Not fine for std::map, std::list, ...

Pointers also don't work for reverse_iterator

- `++ptr` goes forward, but `++` for `reverse_iterator` should go backward

Implementing an iterator

Alternative is to define a new class that handles iterating for Vec

Good example of where *nested* classes are useful:

```
template <typename T>
class Vec {
public:

    class iterator {
        // ...
    };
    class const_iterator {
        // ...
    };
    class reverse_iterator {
        // ...
    };
    ...
};
```

Implementing an iterator

Nested class has access to members of the enclosing class,
including private members

We don't really need that here; each iterator class simply wraps
a layer of operator overloads around a pointer

Implementing an iterator

```
template <typename T>
class Vec {
public:

    class iterator {
        T* it;

    public:
        iterator(T* initial) : it(initial) { }

        // *** Overloads ***
        ...
    };
    ...
};

};
```

Implementing an iterator

What do we need to overload? At least:

- `operator++` for `++it`
- `operator*` for `*it`
- `operator!=` for `while(it != container.end())`

That's all we need for today, but a real-world iterator might additionally handle:

- `operator==`
- `operator->`

Implementing an iterator

```
template <typename T>
class Vec {
public:

    class iterator {
        T* it;

    public:
        iterator(T* initial) : it(initial) { }

        iterator& operator++() { ??? }

        bool operator!=(const iterator& o) const { ??? }

        T& operator*() { ??? }

    };
    ...
};
```

Implementing an iterator

```
template <typename T>
class Vec {
public:

    class iterator {
        T* it;

    public:
        iterator(T* initial) : it(initial) { }

        iterator& operator++() {
            ++it;           // advance pointer 1 slot
            return *this;
        }

        bool operator!=(const iterator& o) const {
            return it != o.it; // if pointers are different, iterators
                               // are pointing to different slots
        }

        T& operator*() {
            return *it;      // dereferencing iterator is
                           // dereferencing pointer
        }
    };
    ...
};
```

Implementing an iterator

Study questions:

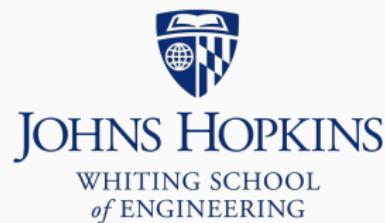
- How would `const_iterator` implementation differ?
 - Hint: `operator*` needs to change
- How would `reverse_iterator` implementation differ?
 - Hint: `operator++` needs to change

auto

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

auto

Use auto in a place where you would otherwise have written a type:

```
for(auto it = vc.begin(); it != vc.end(); it++) {  
    ...  
}
```

Doing so leaves type unspecified; up to the C++ compiler to *infer* appropriate type

auto

Sometimes, this is an easy inference to make:

```
int a = 7;  
auto b = a; // b clearly an int
```

Made slightly trickier by type promotion:

```
int c = 7;  
double d = 11.1;  
auto e = c * d; // e is a double
```

auto

auto variable must be initialized; compiler must be able to infer type immediately

```
auto x = 0;  
// above is OK
```

```
auto y;  
y = 7;  
// compile error:  
// declaration of 'auto y' has no initializer
```

auto

```
#include <iostream>

using std::cout;  using std::endl;

int main() {
    int a = 7;
    auto b = a;
    cout << "b=" << b << ", size=" << sizeof(b) << endl;

    int c = 7;
    double d = 11.1;
    auto e = c * d; // e is a double
    cout << "e=" << e << ", size=" << sizeof(e) << endl;

    return 0;
}
```

auto

```
$ g++ -c auto1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o auto1 auto1.o  
$ ./auto1  
b=7, size=4  
e=77.7, size=8
```

auto

We saw auto in the context of iterators:

```
int sum_every_other(const vector<int>& ls) {
    int total = 0;
    for(vector<int>::const_iterator it = ls.cbegin();
        it != ls.cend(); it++)
    {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

auto

```
int sum_every_other(const vector<int>& ls) {
    int total = 0;
    // using auto instead
    for(auto it = ls.cbegin(); it != ls.cend(); it++) {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

auto

Con: types aren't as clear from the source code

Pro: Source is more concise and, arguably, easier to maintain

- Changes to types on right-hand side propagate to left:

```
int c = 7;  
double d = 11.1;  
// changing d to int would also change e to int!  
auto e = c * d;
```

auto

E.g., changing parameter type from `vector<int>` to `list<int>` automatically propagates to iterator type:

```
int sum_every_other(const list<int>& ls) {
    int total = 0;
    // it *was* vector<int>::const_iterator
    // *now* it's list<int>::const_iterator
    for(auto it = ls.cbegin(); it != ls.cend(); it++) {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

auto

Caution: compiler-inferred type can be unexpected:

```
vector<int> vec = {1, 2, 3};  
const vector<int>& vec_ref;  
auto vec2 = vec_ref;
```

Is vec2 a *copy* of vec, or an alias?

auto

```
#include <iostream>
#include <vector>

using std::cout;  using std::endl;
using std::vector;

int main() {
    vector<int> vec = {1, 2, 3};
    const vector<int>& vec_ref = vec;
    auto vec2 = vec_ref;

    vec[0] = 10;
    cout << "vec[0] = " << vec[0] << endl;
    cout << "vec2[0] = " << vec2[0] << endl;

    return 0;
}
```

auto

```
$ g++ -c auto2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o auto2 auto2.o  
$ ./auto2  
vec[0] = 10  
vec2[0] = 1
```

It's a copy

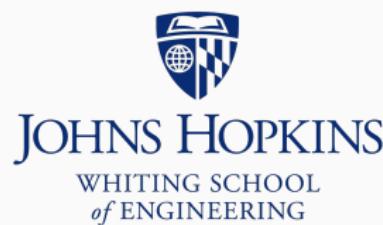
Compiler inferred type `vector<int>` for `vec2`, *not const vector<int>&*

Ranged for

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Ranged for

C++11 gives us a concise loop syntax, similar to Java for-each loops:

```
#include <iostream>

using std::cout; using std::endl;

int main() {
    for(int i : {2, 4, 6, 8}) {
        cout << i << ' ';
    }
    cout << endl;
    return 0;
}
```

Ranged for

```
$ g++ -c ranged_for1.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o ranged_for1 ranged_for1.o  
$ ./ranged_for1  
2 4 6 8
```

Ranged for

Works on our favorite STL containers:

```
#include <iostream>
#include <vector>

using std::cout; using std::endl;
using std::vector;

int main() {
    vector<int> vec = {10, 8, 6, 4, 2, 0};
    for(int i : vec) {
        cout << i << ' ';
    }
    cout << endl;
    return 0;
}
```

Ranged for

```
$ g++ -c ranged_for2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o ranged_for2 ranged_for2.o  
$ ./ranged_for2  
10 8 6 4 2 0
```

Ranged for

Often combined with auto

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

using std::cout; using std::endl;
using std::map;  using std::string;

int main() {
    map<string, int> pres;
    pres["Washington"] = 1789;
    pres["Adams"] = 1797;
    pres["Jefferson"] = 1801;

    // compiler infers pair<string, int>
    for(auto kv : pres) {
        cout << kv.first << ':' << kv.second << endl;
    }
    return 0;
}
```

Ranged for

```
$ g++ -c ranged_for3.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o ranged_for3 ranged_for3.o  
$ ./ranged_for3  
Adams:1797  
Jefferson:1801  
Washington:1789
```

Avoided writing either the iterator type or the type of the dereferenced iterator

- Iterator type: `map<string, int>::iterator`
- Dereferenced iterator type: `pair<string, int>`

Ranged for

Works with pretty much any data structure with begin() & end()
returning appropriate iterators

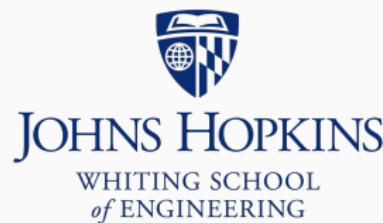
Including all STL containers we've discussed in class

override

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

override

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

using std::cout; using std::endl;

class Account {
public:
    virtual std::string type() const { return "Account"; }
};

class CheckingAccount : public Account {
public:
    virtual std::string type()          { return "CheckingAccount"; }
};

int main() {
    CheckingAccount checking;
    Account& acct = checking;
    cout << acct.type() << endl; // polymorphism FTW?
    return 0;
}
```

override

```
$ g++ -c override.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o override override.o  
$ ./override  
Account
```

Sometimes you *intend* to override a function in the base class...
...but you fail

override

In this case, it was just a matter of missing a const

```
class Account {  
public:  
    virtual std::string type() const { return "Account"; }  
};  
class CheckingAccount : public Account {  
public:  
    virtual std::string type() { return "CheckingAccount"; }  
    // ^^^^^^ oops  
};
```

override

```
$ g++ -c override_fix.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o override_fix override_fix.o  
$ ./override_fix  
CheckingAccount
```

That's better

This is a typical mistake, often because we:

- fail to match const status
- fail to exactly match parameter & return types

override

The `override` keyword helps

When you *intend* to override a function, add the `override` modifier:

```
class Account {  
public:  
    virtual std::string type() const { return "Account"; }  
};  
class CheckingAccount : public Account {  
public:  
    virtual std::string type()           override { return "CheckingAccount"; }  
    //                                     ^^^^^^ oops  
};
```

override

```
$ g++ -c override_fix2.cpp -std=c++11 -pedantic -Wall -Wextra
override_fix2.cpp:12:25: error: 'virtual std::__cxx11::string
CheckingAccount::type()' marked 'override', but does not override
    virtual std::string type() override { return "CheckingAccount"; }
                           ^~~~
```

override

Now we combine it with const to fix the problem:

```
class Account {  
public:  
    virtual std::string type() const { return "Account"; }  
};  
class CheckingAccount : public Account {  
public:  
    virtual std::string type() const override { return "CheckingAccount"; }  
};
```

```
$ g++ -c override_fix2.cpp -std=c++11 -pedantic -Wall -Wextra  
$ g++ -o override_fix2 override_fix2.o  
$ ./override_fix2  
CheckingAccount
```

No warnings or errors because override was successful