

## Course Outline

### 1. Overview of Memory Types

1. Memory Addresses and Hexadecimal Numbers
2. Using the Debugger to Analyze Memory
3. Types of Computer Memory
4. Cache Memory
5. Virtual Memory

### 2. Variables and Memory

1. The Process Memory Model
2. Automatic Memory Allocation (The Stack)
3. Call-By-Value vs. Call-By-Reference

### 3. Dynamic Memory Allocation (The Heap)

1. Heap Memory
2. Using malloc and free
3. Using new and delete
4. Typical Memory Management Problems

### 4. Resource Copying Policies

1. Copy Semantics
2. Lvalues and rvalues
3. Move Semantics



## 5. Smart Pointers

1. Resource Acquisition Is Initialization (RAII)
2. Smart pointers
3. Transferring ownership

## 6. Project: Memory Management Chatbot

## Memory Addresses and Hexadecimal Numbers

Understanding the number system used by computers to store and process data is essential for effective memory management, which is why we will start with an introduction into the binary and hexadecimal number systems and the structure of memory addresses.

Early attempts to invent an electronic computing device met with disappointing results as long as engineers and computer scientists tried to use the decimal system. One of the biggest problems was the low distinctiveness of the individual symbols in the presence of [noise](#). A 'symbol' in our alphabet might be a letter in the range A-Z while in our decimal system it might be a number in the range 0-9. The more symbols there are, the harder it can be to differentiate between them, especially when there is electrical interference. After many years of research, an early pioneer in computing, John Atanasoff, proposed to use a coding system that expressed numbers as sequences of only two digits: one by the presence of a charge and one by the absence of a charge. This numbering system is called Base 2 or binary and it is represented by the digits 0 and 1 (called 'bit') instead of 0-9 as with the decimal system. Differentiating between only two symbols, especially at high frequencies, was much easier and more robust than with 10 digits. In a way, the ones and zeroes of the binary system can be compared to Morse Code, which is also a very robust way to transmit information in the presence of much interference. This was one of the primary reasons why the binary system quickly became the standard for computing.

Inside each computer, all numbers, characters, commands and every imaginable type of information are represented in binary form. Over the years, many coding schemes and techniques were invented to manipulate these 0s and 1s effectively. One of the most widely used schemes is called ASCII (*American Standard Code for Information Interchange*), which lists the binary code for a set of 127 characters. The idea was to represent each letter with a sequence of binary numbers so that storing texts on in computer memory and on hard (or floppy) disks would be possible.

The film enthusiasts among you might know the scene in the hit movie "The Martian" with Matt Damon, in which an ASCII table plays an important role in the rescue from Mars.

The following figure shows an ASCII table, where each character (rightmost column) is associated with an 8-digit binary number:



The letter **U** for example can be represented by the following sequence of bits: **0101 0101**

QUESTION 1 OF 2

Can you figure out the binary sequence for the word "UDACITY"?

- 01001000 01100101 01101100 01101100 01101111 0100000 01010111 01101111  
01110010 01101100 01100100
- 01110100 01110010 01111001 01000001 01100111 01100001 01101001 01101110
- 01010101 01000100 01000001 01000011 01001001 01010100 01011001
- 01010101 01000100 01001111 01001110 01001101 01001101 01001101

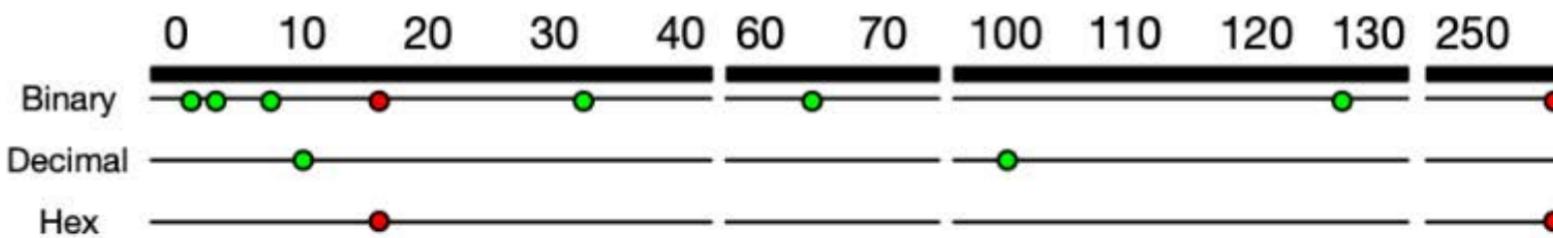
SUBMIT

In addition to the decimal number (column "Dec") and the binary number, the ASCII table provides a third number for each character (column "Hex"). According to the table above, the letter **z** is referenced by the decimal number **122**, by the binary number **0111 1010** and by **7A**. You have probably seen this type of notation before, which is called "*hexadecimal*". Hexadecimal (hex) numbers are used often in computer systems, e.g for displaying memory readouts - which is why we will look into this topic a little bit deeper. Instead of having a base of 2 (such as binary numbers) or a base of 10 (such as our conventional decimal numbers), hex numbers have a base of 16. The conversion between the different numbering systems is a straightforward operation and can be easily performed with any scientific calculator. More details on how to do this can e.g. be found [here](#).

There are several reasons why it is preferable to use hex numbers instead of binary numbers (which computers store at the lowest level), three of which are given below:

- 1. Readability:** It is significantly easier for a human to understand hex numbers as they resemble the decimal numbers we are used to. It is simply not intuitive to look at binary numbers and decide how big they are and how they relate to another binary number.
- 2. Information density:** A hex number with two digits can express any number from 0 to 255 (because  $16^2$  is 256). To do the same in the binary system, we would require 8 digits. This difference is even more pronounced as numbers get larger and thus harder to deal with.
- 3. Conversion into bytes:** Bytes are units of information consisting of 8 bits. Almost all computers are byte-addressed, meaning all memory is referenced by byte, instead of by bit. Therefore, using a counting system that can easily convert into bytes is an important requirement. We will shortly see why grouping bits into a byte plays a central role in understanding how computer memory works.

The reason why early computer scientists have decided to not use decimal numbers can also be seen in the figure below. In these days (before pocket calculators were widely available), programmers had to interpret computer output in their head on a regular basis. For them, it was much easier and quicker to look at and interpret **7E** instead of **0111 1110**. Ideally, they would have used the decimal system, but the conversion between base 2 and base 10 is much harder than between base 2 and base 16. Note in the figure that the decimal system's digit transitions never match those of the binary system. With the hexadecimal system, which is based on a multiple of 2, digit transitions match up each time, thus making it much easier to convert quickly between these numbering systems.



Each dot represents an increase in the number of digits required to express a number in different number systems. For base 2, this happens at 2, 4, 8, 32, 64, 128 and 256. The red dots indicate positions where several numbering systems align. Note that there are breaks in the number line to conserve space.

QUESTION 2 OF 2

Convert the following numbers from binary into hex and vice-versa:

*Submit to check your answer choices!*

NUMBER

CONVERSION

FF

1111 1111

1D

11101

1101 1001

D9

1001 0110

96



# ND213 C03 L01 02.2 Using The Debugger To Analyze Memory SC

```

1 #include <stdio.h>
2
3 int main()
4 {
5     char str1[] = "UDACITY";
6     printf("%s", str1);
7
8     return 0;
9 }
```

PROBLEMS OUTPUT TERMINAL ...

3: gdb

```

imac-pro:src ahaja$ gdb ./bin/example
GNU gdb (GDB) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin18.5.0".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bin/example...
Reading symbols from /Users/ahaja/Dropbox/Nebentaetigkeit/Seminare/Udacity/Courses/C++ ND/
Memory Management Course/Code/Examples/bin/example.dSYM/Contents/Resources/DWARF/example
(gdb) 
```

## GDB cheatsheet - page 1

<b>Running</b>	<b>&lt;where&gt;</b>
# gdb <program> [core dump] Start GDB (with optional core dump).	function_name Break/watch the named function.
# gdb --args <program> <args...> Start GDB and pass arguments	line_number Break/watch the line number in the current source file.
# gdb --pid <pid> Start GDB and attach to process.	file:line_number Break/watch the line number in the named source file.
set args <args...> Set arguments to pass to program to be debugged.	
run Run the program to be debugged.	
kill Kill the running program.	
<b>Breakpoints</b>	
break <where> Set a new breakpoint.	
delete <breakpoint#> Remove a breakpoint.	
clear Delete all breakpoints.	
enable <breakpoint#> Enable a disabled breakpoint.	
disable <breakpoint#> Disable a breakpoint.	
<b>Watchpoints</b>	
watch <where> Set a new watchpoint.	
delete/enable/disable <watchpoint#> Like breakpoints.	
<b>Conditions</b>	
break/watch <where> if <condition> Break/watch at the given location if the condition is met.	
Conditions may be almost any C expression that evaluate to true or false.	
condition <breakpoint#> <condition> Set/change the condition of an existing break- or watchpoint.	
<b>Examining the stack</b>	
backtrace where Show call stack.	
backtrace full where full Show call stack, also print the local variables in each frame.	
frame <frame#> Select the stack frame to operate on.	
<b>Stepping</b>	
step Go to next instruction (source line), diving into function.	

© 2007 Marc Haisenko <marc@darkdust.net>

## GDB cheatsheet - page 2

<b>Format</b>	<b>Manipulating the program</b>	<b>Informations</b>
a Pointer.	set var <variable_name><value> Change the content of a variable to the given value.	disasemble disasemble <where> Disasembles the current function or given location.
c Read as integer, print as character.	return <expression> From the current function to return.	info args Print the arguments to the function of the current stack frame.
d Integer, signed decimal.		info breakpoints Print informations about the break- and watchpoints.
f Floating point number.		
e Integer, print as octal.		
x Integer, print as hexadecimal.		
<b>&lt;what&gt;</b>		
expression		

So let's fire up this program or fire up GDB using this program.

ND213 C03 L01 02.2 Using The Debugger To Analyze Memory SC

**GDB cheatsheet - page 1**

<b>Running</b> <pre># gdb &lt;program&gt; [core dump] # gdb --args &lt;program&gt; &lt;args... # gdb --pid &lt;pid&gt; set args &lt;args... run kill</pre>	<b>&lt;where&gt;</b> <pre>function_name line_number file:line_number</pre>	<b>next</b> Go to next instruction (source line) but don't dive into functions.
<b>Conditions</b> <pre>break/watch &lt;where&gt; if &lt;condition&gt; condition &lt;breakpoint#&gt; &lt;condition&gt;</pre>	<b>finish</b> Continue until the current function returns.	<b>Variables and memory</b> <pre>print/format &lt;what&gt; display/format &lt;what&gt; undisplay &lt;display#&gt; enable display &lt;display#&gt; disable display &lt;display#&gt; x/nfu &lt;address&gt;</pre>
<b>Breakpoints</b> <pre>break &lt;where&gt; delete &lt;breakpoint#&gt; clear enable &lt;breakpoint#&gt; disable &lt;breakpoint#&gt;</pre>	<b>Examining the stack</b> <pre>backtrace where backtrace full where full frame &lt;frame#&gt;</pre>	<b>Print content of variable/memory location/register.</b> <b>Like „print“, but print the information after each stepping instruction.</b> <b>Remove the „display“ with the given number.</b> <b>En- or disable the „display“ with the given number.</b> <b>Print memory.</b> <b>n: How many units to print (default 1).</b> <b>f: Format character (like „print“).</b> <b>u: Unit.</b> <b>Unit is one of:</b> <b>b: Byte,</b> <b>h: Half-word (two bytes)</b> <b>w: Word (four bytes)</b> <b>g: Giant word (eight bytes).</b>
<b>Watchpoints</b> <pre>watch &lt;where&gt; delete/enable/disable &lt;watchpoint#&gt;</pre>	<b>Stepping</b> <pre>step</pre>	<b>© 2007 Marc Haisenko &lt;marc@darkdust.net&gt;</b>

**GDB cheatsheet - page 2**

<b>Format</b> <pre>a c d f o x</pre>	<b>Manipulating the program</b> <pre>set var &lt;variable_name&gt;=&lt;value&gt; return &lt;expression&gt;</pre>	<b>Informations</b> <pre>disassemble disassemble &lt;where&gt; info breakpoints info breakpoints &lt;function&gt; info sources info directory &lt;directory&gt; info expression &lt;expression&gt;</pre>
<b>&lt;what&gt;</b> <pre>expression</pre>	<b>Sources</b> <pre>list list &lt;filename&gt;:&lt;function&gt; list &lt;filename&gt;:&lt;line_number&gt; list &lt;function&gt;::&lt;class&gt;</pre>	<b>Prints to the function of the current frame.</b> <b>Prints to the function of the current frame.</b> <b>Print informations about the break- and watchpoints.</b> <b>Prints the local variables in the current frame.</b>

we have to actually run the program which can be simply done by typing run.

ND213 C03 L01 02.2 Using The Debugger To Analyze Memory SC

GDB cheatsheet - page 1

<b>Running</b>	<b>&lt;where&gt;</b>	next
# gdb <program> [core dump] Start GDB (with optional core dump).	function_name Break/watch the named function.	Go to next instruction (source line) but don't dive into functions.
# gdb --args <program> <args...> Start GDB and pass arguments	line_number Break/watch the line number in the current source file.	finish
# gdb --pid <pid> Start GDB and attach to process.	file:line_number Break/watch the line number in the named source file.	Continue until the current function returns.
set args <args...> Set arguments to pass to program to be debugged.		continue
run Run the program to be debugged.		Continue normal execution.
kill Kill the running program.		

**Conditions**

break/watch <where> if <condition>  
Break/watch at the given location if the condition is met.  
Conditions may be almost any C expression that evaluate to true or false.

condition <breakpoint#> <condition>  
Set/change the condition of an existing break- or watchpoint.

**Breakpoints**

break <where>  
Set a new breakpoint.

delete <breakpoint#>  
Remove a breakpoint.

clear  
Delete all breakpoints.

enable <breakpoint#>  
Enable a disabled breakpoint.

disable <breakpoint#>  
Disable a breakpoint.

**Watchpoints**

watch <where>  
Set a new watchpoint.

delete/enable/disable <watchpoint#>  
Like breakpoints.

**Examining the stack**

backtrace  
where  
Show call stack.

backtrace full  
where full  
Show call stack, also print the local variables in each frame.

frame <frame#>  
Select the stack frame to operate on.

**Stepping**

step  
Go to next instruction (source line), diving into function.

© 2007 Marc Haisenko <marc@darkdust.net>

GDB cheatsheet - page 2

<b>Format</b>	<b>Manipulating the program</b>	<b>Informations</b>
a Pointer.	set var <variable_name>=<value> Change the content of a variable to the given value.	disassemble disassemble <where> Disassemble the current function or given location.
c Read as integer, print as character.	return <expression> Force the current function to return immediate value.	info args Print the arguments to the function of the current stack frame.
d Integer, signed decimal.		info breakpoints Print informations about the break- and watchpoints.
f Floating point number.		info display Shows about the .obj files.
o Integer, print as octal		info locals Shows the local variables in the current frame.
x Integer, print as hexadecimal.		list list <filename>:<function> list <filename>:<line_number> list <filename>:<label>
	<what>	info sources Shows the current sources.
		list offsets <label>
		info display
		CC
		HD YouTube
		Info locals

the A, and so on and so forth are located in memory.

ND213 C03 L01 02.3 Using The Debugger To Analyze Memory SC

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char str1[] = "UDACITY";
6     printf("%s", str1);
7
8     return 0;
9 }
```

PROBLEMS OUTPUT TERMINAL ...

```
3: gdb
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc_darwin/install/TmpContent/Objects/Libc.build/libsystem_darwin.dylib.build/Objects-normal/x86_64/mach.o': can't open to read symbols: No such file or directory.
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc_darwin/install/TmpContent/Objects/Libc.build/libsystem_darwin.dylib.build/Objects-normal/x86_64/stdio.o': can't open to read symbols: No such file or directory.
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc_darwin/install/TmpContent/Objects/Libc.build/libsystem_darwin.dylib.build/Objects-normal/x86_64/stdc硐.o': can't open to read symbols: No such file or directory.
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc_darwin/install/TmpContent/Objects/Libc.build/libsystem_darwin.dylib.build/Objects-normal/x86_64/string.o': can't open to read symbols: No such file or directory.
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc_darwin/install/TmpContent/Objects/Libc.build/libsystem_darwin.dylib.build/Objects-normal/x86_64/variant.o': can't open to read symbols: No such file or directory.

Thread 3 hit Breakpoint 1, main ()
at /Users/ahaja/Dropbox/Nebentaeigkeit/Seminare/Udacity/Courses/C++ ND/Memory Management Course/Code/Examples/src/example.cpp:5
5     char s
(gdb) step
6     printf("%s", str1)
(gdb) p str1
$1 = "UDACITY"
(gdb) p &str1
```

1:22 / 3:30

### GDB cheatsheet - page 1

<b>Running</b>	<b>&lt;where&gt;</b>	next	
# gdb <program> [core dump]	function_name	Go to next instruction (source line) but don't dive into functions.	
Start GDB (with optional core dump).	Break/watch the named function.		
# gdb --args <program> <args...>	line_number	finish	
Start GDB and pass arguments	Break/watch the line number in the current source file.	Continue until the current function returns.	
# gdb --pid <pid>	file:line_number	continue	
Start GDB and attach to process.	Break/watch the line number in the named source file.	Continue normal execution.	
set args <args...>			
Set arguments to pass to program to be debugged.			
run			
Run the program to be debugged.			
kill			
Kill the running program.			
<b>Breakpoints</b>			
break <where>	Set a new breakpoint.		
delete <breakpoint#>	Remove a breakpoint.		
clear	Delete all breakpoints.		
enable <breakpoint#>	Enable a disabled breakpoint.		
disable <breakpoint#>	Disable a breakpoint.		
<b>Watchpoints</b>			
watch <where>	Set a new watchpoint.		
delete/enable/disable <watchpoint#>	Like breakpoints.		
<b>Examining the stack</b>			
backtrace	where	enable display <display#>	
where	Show call stack.	disable display <display#>	
backtrace full		En- or disable the „display“ with the given number.	
where full	Show call stack, also print the local variables in each frame.	x/nfu <address>	
frame <frame#>	Select the stack frame to operate on.	Print memory.	
<b>Stepping</b>			
step	Go to next instruction (source line), diving into function.	n: How many units to print (default 1).	
			f: Format character (like „print“).
			u: Unit.
			Unit is one of:
			b: Byte,
			h: Half-word (two bytes)
			w: Word (four bytes)
			g: Giant word (eight bytes).

© 2007 Marc Haisenko <marc@darkdust.net>

### GDB cheatsheet - page 2

<b>Format</b>	<b>Manipulating the program</b>	<b>Informations</b>
a Pointer.	set var <variable_name>=<value>	disasemble
c Read as integer, print as character.	Change the content of a variable to the given value.	disasemble <where>
d Integer, signed decimal.	return <expression>	Disasemble the current function or given location.
f Floating point number.	Print the current value to return.	info args
o Integer, print as octal		Print the arguments to the function of current stack frame.
x Integer, print as hexadecimal.		info breakpoints
<b>&lt;what&gt;</b>		
Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).		
<b>SOURCES</b>		
directory <directory>	Add directory to the list of directories that is searched for sources.	info display
		Print informations about the break- and watchpoints.
list		HD YouTube
list <filename>:<function>		info locals
list <filename>:<line_number>		Get the local variables in the current
list <filename>:<class>		

So the unit we are going to print seven successive bytes in binary form.

ND213 C03 L01 02.3 Using The Debugger To Analyze Memory SC

**GDB cheatsheet - page 1**

<p><b>Running</b></p> <pre># gdb &lt;program&gt; [core dump] # gdb --args &lt;program&gt; &lt;args... # gdb --pid &lt;pid&gt; set args &lt;args... run kill</pre> <p><b>Breakpoints</b></p> <pre>break &lt;where&gt; delete &lt;breakpoint#&gt; clear enable &lt;breakpoint#&gt; disable &lt;breakpoint#&gt;</pre> <p><b>Watchpoints</b></p> <pre>watch &lt;where&gt; delete/enable/disable &lt;watchpoint#&gt;</pre>	<p><b>&lt;where&gt;</b></p> <pre>function_name line_number file:line_number</pre> <p><b>Conditions</b></p> <pre>break/watch &lt;where&gt; if &lt;condition&gt; condition &lt;breakpoint#&gt; &lt;condition&gt;</pre> <p><b>Examining the stack</b></p> <pre>backtrace where backtrace full where full frame &lt;frame#&gt;</pre> <p><b>Stepping</b></p> <pre>step</pre>	<p><b>Variables and memory</b></p> <pre>print/format &lt;what&gt; display/format &lt;what&gt; undisplay &lt;display#&gt; enable display &lt;display#&gt; disable display &lt;display#&gt; x/nfu &lt;address&gt;</pre> <p><b>Unit</b></p> <ul style="list-style-type: none"> <li><i>b</i>: Byte,</li> <li><i>h</i>: Half-word (two bytes)</li> <li><i>w</i>: Word (four bytes)</li> <li><i>g</i>: Giant word (eight bytes).</li> </ul> <p><b>Information</b></p> <pre>disassemble info args info breakpoints</pre> <p><b>Expressions</b></p> <pre>expression &lt;what&gt;</pre>
---	---	--

next

Go to next instruction (source line) but don't dive into functions.

finish

Continue until the current function returns.

continue

Continue normal execution.

Print content of variable/memory location/register.

Like „print“, but print the information after each stepping instruction.

Remove the „display“ with the given number.

Print memory.

*n*: How many units to print (default 1).

*f*: Format character (like „print“).

*u*: Unit.

Unit is one of:

- b*: Byte,
- h*: Half-word (two bytes)
- w*: Word (four bytes)
- g*: Giant word (eight bytes).

© 2007 Marc Haisenko <marc@darkdust.net>

**GDB cheatsheet - page 2**

<p><b>Format</b></p> <pre>a c d f o</pre> <p><b>Manipulating the program</b></p> <pre>set var &lt;variable_name&gt;=&lt;value&gt; return &lt;expression&gt;</pre> <p><b>Informations</b></p> <pre>disassemble info args info breakpoints</pre>	<p><b>Manipulating the program</b></p> <pre>set var &lt;variable_name&gt;=&lt;value&gt; return &lt;expression&gt;</pre> <p><b>Informations</b></p> <pre>disassemble &lt;where&gt; info args info breakpoints</pre>
--	--

Change the content of a variable to the given value.

Focus the current function to return immediate value.

Add directory to the list of directories that GDB should search for source files.

Print the arguments to the function of the current stack frame.

Print informations about the break- and watchpoints.

list

Almost any C expression, including function calls (must be prefixed with a cast to tell GDB the return value type).

list <filename>:<function>
list <filename>:<line\_number>
list <filename>:<class>

info display

info locals

info display

CC

HD

YouTube

Sources

Copyright © 2007 Marc Haisenko

PROBLEMS OUTPUT TERMINAL ... 3: gdb

n to read symbols: No such file or directory.  
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc\_darwin/install/TmpContent /Objects/Libc.build/libsystem\_darwin.dylib.build/Objects-normal/x86\_64/stdlib.o': can't open to read symbols: No such file or directory.  
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc\_darwin/install/TmpContent /Objects/Libc.build/libsystem\_darwin.dylib.build/Objects-normal/x86\_64/string.o': can't open to read symbols: No such file or directory.  
warning: `/BuildRoot/Library/Caches/com.apple.xbs/Binaries/Libc\_darwin/install/TmpContent /Objects/Libc.build/libsystem\_darwin.dylib.build/Objects-normal/x86\_64/variant.o': can't open to read symbols: No such file or directory.

Thread 3 hit Breakpoint 1, main ()  
at /Users/ahaja/Dropbox/Nebentaetigkeit/Seminare/Udacity/Courses/C++ ND/Memory Management Course/Code/Examples/src/example.cpp:5  
5 char str1[] = "UDACITY";  
(gdb) step  
6 printf("%s", str1);  
(gdb) p str1  
\$1 = "UDACITY"  
(gdb) p &str1  
\$2 = (char (\*)[8]) 0x7fffebf940  
(gdb) x/7tb 0x7fffebf940  
0x7fffebf940: 01010101 01000100 01000001 01000011 01001001 0  
(gdb) x/7xb 0x7fffebf940  
0x7fffebf940: 0x41 0x43 0x49 0x54 0x59  
(gdb)

you might also be able to track down certain errors.

## Using the Debugger to Analyze Memory

As you have seen in the last section, binary numbers and hex numbers can be used to represent information. A coding scheme such as an ASCII table makes it possible to convert text into binary form. In the following, we will try to look at computer memory and locate information there.

In the following example, we will use the debugger to look for a particular string in computer memory. Depending on your computer operating system and on the compiler you have installed, there might be several debugging tools available to you. In the following video, we will use the gdb debugger to locate the character sequence "UDACITY" in computer memory. The code below creates an array of characters in computer memory (on the stack, which we will learn more about shortly) and prints it to the console:

```
#include <stdio.h>

int main()
{
    char str1[] = "UDACITY";
    printf("%s",str1);

    return 0;
}
```

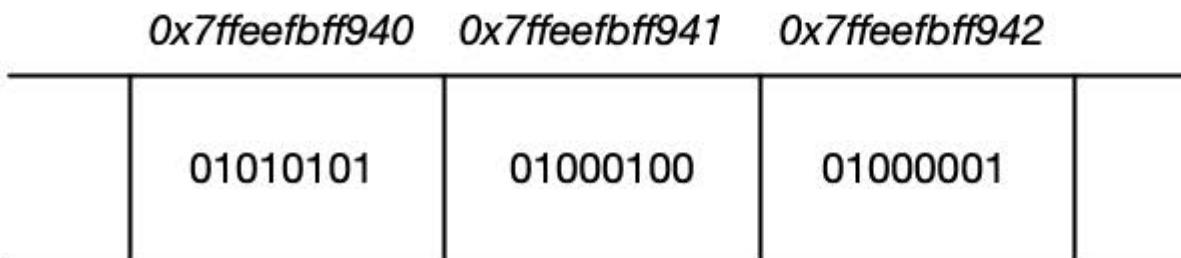
Let us try to locate the string in memory using gdb.

As you have just seen in the video, the binary ASCII codes for the letters in UDACITY could be located in computer memory by using the address of the variable `str1` from the code example above. The output of gdb can also be observed in the following image:

```
(gdb) p str1
$1 = "UDACITY"
(gdb) p &str1
$2 = (char (*)[8]) 0x7fffeefbff940
(gdb) x/7tb 0x7fffeefbff940
0x7fffeefbff940: 01010101      01000100      01000001      01000011      01001001      01010100      01011001
(gdb) x/7xb 0x7fffeefbff940
0x7fffeefbff940: 0x55 0x44 0x41 0x43 0x49 0x54 0x59
```

You can clearly see that using hex numbers to display the information is a much shorter and more convenient form for a human programmer than looking at the binary numbers. Note that hex numbers are usually prepended with "0x".

Computer memory is treated as a sequence of cells. This means that we can use the starting address to retrieve the byte of information stored there. The following figure illustrates the principle:



Computer memory represented as a sequence of data cells (e.g. 01010101) with their respective memory addresses shown on top.

Let us perform a short experiment using gdb again: By adding 1, 2, 3, ... to the address of the string variable `str1`, we can proceed to the next cell until we reach the end of the memory we want to look at.

```
(gdb) x/1xb 0x7fffeefbff940
0x7fffeefbff940: 0x55
(gdb) x/1xb 0x7fffeefbff941
0x7fffeefbff941: 0x44
(gdb) x/1xb 0x7fffeefbff942
0x7fffeefbff942: 0x41
(gdb) x/1xb 0x7fffeefbff943
0x7fffeefbff943: 0x43
(gdb) x/1xb 0x7fffeefbff944
0x7fffeefbff944: 0x49
(gdb) x/1xb 0x7fffeefbff945
0x7fffeefbff945: 0x54
(gdb) x/1xb 0x7fffeefbff946
0x7fffeefbff946: 0x59
(gdb) x/1xb 0x7fffeefbff947
0x7fffeefbff947: 0x00
```

Note that the numbers above represent the string "UDACITY" again. Also note that once we exceed the end of the string, the memory cell has the value 0x00. This means that the experiment has shown that an offset of 1 in a hexadecimal address corresponds to an offset of 8 bits (or 1 byte) in computer memory.

## Your Turn

Unfortunately, gdb will not work in Udacity Workspaces, but you can still try this exercise either in your local environment if you have g++ and gdb installed, or you can use [OnlineGDB](#) to follow along.

Try to locate the characters of "Udacity" using gdb in your local environment or in the online debugger.

### Using GDB Locally

In order to use gdb locally, you will need to compile `main.cpp` with **debugging symbols**. This can be done with the `-g` option for g++:

```
g++ -g main.cpp
```

You can then run gdb on the output with:

```
gdb a.out
```

When gdb displays the line `Type <RET> for more, q to quit, c to continue without paging`, be sure to press the RETURN key to continue.

### Using GDB Online

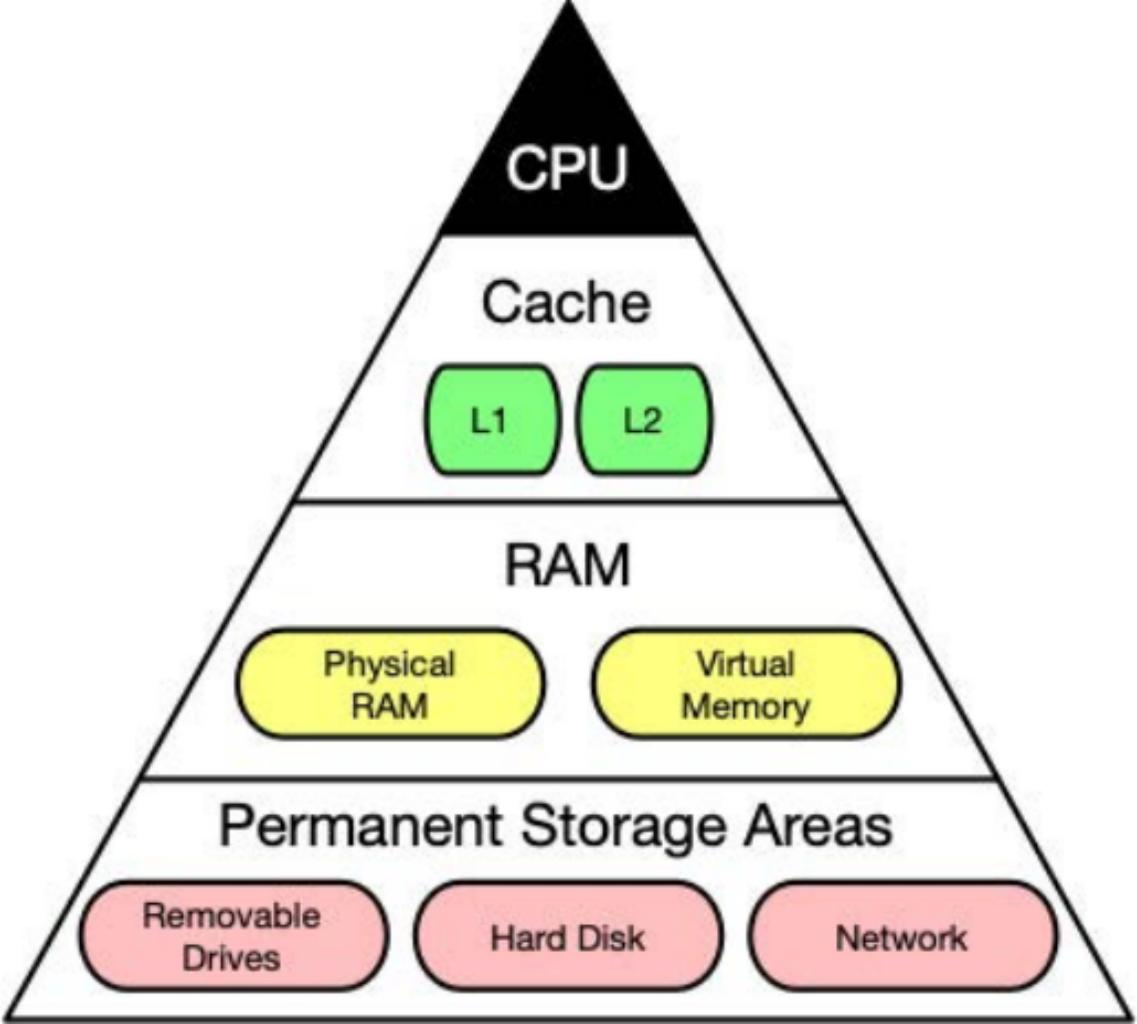
To use the OnlineGDB application, simply paste the code into the online editor and press the "Debug" button at the top of the screen.

In a course on memory management we obviously need to take a look at the available memory types in computer systems. Below you will find a small list of some common memory types that you will surely have heard of:

- RAM / ROM
- Cache (L1, L2)
- Registers
- Virtual Memory
- Hard Disks, USB drives

Let us look into these types more deeply: When the CPU of a computer needs to access memory, it wants to do this with minimal latency. Also, as large amounts of information need to be processed, the available memory should be sufficiently large with regard to the tasks we want to accomplish.

Regrettably though, low latency and large memory are not compatible with each other (at least not at a reasonable price). In practice, the decision for low latency usually results in a reduction of the available storage capacity (and vice versa). This is the reason why a computer has multiple memory types that are arranged hierarchically. The following pyramid illustrates the principle:



As you can see, the CPU and its ultra-fast (but small) registers used for short-term data storage reside at the top of the pyramid. Below are Cache and RAM, which belong to the category of temporary memory which quickly loses its content once power is cut off. Finally, there are permanent storage devices such as the ROM, hard drives as well as removable drives such as USB sticks.

Let us take a look at a typical computer usage scenario to see how the different types of memory are used:

1. After switching on the computer, it loads data from its read-only memory (ROM) and performs a power-on self-test (POST) to ensure that all major components are working properly. Additionally, the computer memory controller checks all of the memory addresses with a simple read/write operation to ensure that memory is functioning correctly.
2. After performing the self-test, the computer loads the basic input/output system (BIOS) from ROM. The major task of the BIOS is to make the computer functional by providing basic information about such things as storage devices, boot sequence, security or auto device recognition capability.
3. The process of activating a more complex system on a simple system is called "bootstrapping": It is a solution for the chicken-egg-problem of starting a software-driven system by itself using software. During bootstrapping, the computer loads the operating system (OS) from the hard drive into random access memory (RAM). RAM is considered "random access" because any memory cell can be accessed directly by intersecting the respective row and column in the matrix-like memory layout. For performance reasons, many parts of the OS are kept in RAM as long as the computer is powered on.

4. When an application is started, it is loaded into RAM. However, several application components are only loaded into RAM on demand to preserve memory. Files that are opened during runtime are also loaded into RAM. When a file is saved, it is written to the specified storage device. After closing the application, it is deleted from RAM.

This simple usage scenario shows the central importance of the RAM. Every time data is loaded or a file is opened, it is placed into this temporary storage area - but what about the other memory types above the RAM layer in the pyramid?

To maximize CPU performance, fast access to large amounts of data is critical. If the CPU cannot get the data it needs, it stops and waits for data availability. Thus, when designing new memory chips, engineers must adapt to the speed of the available CPUs. The problem they are facing is that memory which is able to keep up with modern CPUs running at several GHz is extremely expensive. To combat this, computer designers have created the memory tier system which has already been shown in the pyramid diagram above. The solution is to use expensive memory in small quantities and then back it up using larger quantities of less expensive memory.

The cheapest form of memory available today is the hard disk. It provides large quantities of inexpensive and permanent storage. The problem of a hard disk is its comparatively low speed - even though access times with modern solid state disks (SSD) have decreased significantly compared to older magnetic-disc models.

The next hierarchical level above hard disks or other external storage devices is the RAM. We will not discuss in detail how it works but only take a look at some key performance metrics of the CPU at this point, which place certain performance expectations on the RAM and its designers:

1. The **bit size** of the CPU decides how many bytes of data it can access in RAM memory at the same time. A 16-bit CPU can access 2 bytes (with each byte consisting of 8 bit) while a 64-bit CPU can access 8 bytes at a time.
2. The **processing speed** of the CPU is measured in Gigahertz or Megahertz and denotes the number of operations it can perform in one second.

From processing speed and bit size, the data rate required to keep the CPU busy can easily be computed by multiplying bit size with processing speed. With modern CPUs and ever-increasing speeds, the available RAM in the market will not be fast enough to match the CPU data rate requirements.

**There are two major principles with caches which are**



0:34 / 0:58



YouTube



called spatial and temporal locality,



0:37 / 0:58



YouTube



ND213 C03 L01 04.2 Cache Memory SC

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int vfeizellcivel;
9     typedef std::chrono::time_point<...> std::chrono::steady_clock::time_point
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[j][i] = i + j;
16            std::cout << &x[j][i] << ": i=" << i << ", j=" << j << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }
```

CALL STACK

WATCH

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 5: BUILD and LAUNCH MM-+

bash-3.2\$

you can now see it in the pop up here and by using

ND213 C03 L01 04.2 Cache Memory SC

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[j][i] = i + j;
16            std::cout << &x[j][i] << ": i=" << i << ", j=" << j << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }
```

0x100002114: i=1, j=1  
0x100002124: i=1, j=2  
0x100002134: i=1, j=3  
0x100002108: i=2, j=0  
0x100002118: i=2, j=1  
0x100002128: i=2, j=2  
0x100002138: i=2, j=3  
0x10000211c: i=3, j=0  
0x10000211c: i=3, j=1  
0x10000212c: i=3, j=2  
0x10000213c: i=3, j=3

Execution time: 98 microseconds

actually is not the time it takes to access the array.

The screenshot shows a video player interface with a C++ code editor and a terminal window below it.

**Code Editor:**

```
ND213 C03 L01 04.2 Cache Memory SC
example.cpp x
example.cpp > ...
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[j][i] = i + j;
16            std::cout << &x[j][i] << ": i=" << i << ", j=" << j << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }
```

**Terminal Output:**

```
0x100002114: i=1, j=1
0x100002124: i=1, j=2
0x100002134: i=1, j=3
0x100002108: i=2, j=0
0x100002118: i=2, j=1
0x10000210c: i=3, j=0
0x10000211c: i=3, j=1
0x10000212c: i=3, j=2
0x10000213c: i=3, j=3
Execution time: 98 microseconds
```

**Annotations:**

A large white box highlights the terminal output area, containing the text: "This is the amount of time it takes to print something to".

**Video Player Controls:**

Watch later Share

4:15 / 5:36

ND213 C03 L01 04.2 Cache Memory SC

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[j][i] = i + j;
16            std::cout << &x[j][i] << ":" << i << ", " << j << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }
```

the console which is a very slow operation and now as to see what I mean,

0x100002114: i=1, j=1  
0x100002124: i=1, j=2  
0x100002134: i=1, j=3  
0x100002108: i=2, j=0  
0x100002118: i=2, j=1  
0x10000210c: i=3, j=0  
0x10000211c: i=3, j=1  
0x10000212c: i=3, j=2  
0x10000213c: i=3, j=3

Execution time: 98 microseconds

The screenshot shows a video player interface with a C++ code editor and a terminal window.

**Code Editor:**

- File: example.cpp
- Content:

```
ND213 C03 L01 04.2 Cache Memory SC
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[j][i] = i + j;
16            //std::cout << &x[j][i] << ": i=" << i << ", j=" << j << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " microseconds" << std::endl;
24
25    return 0;
26 }
```

**Terminal Output:**

```
Execution time: 0 microseconds
bash-3.2$
```

**Bottom Text:**

It's even below the scale we can measure using the microseconds.

**Video Player Controls:**

- Play/Pause
- Captions: on the screen
- Volume: 4:27 / 5:36
- C++: on catch
- CC
- HD
- YouTube
- Share

The screenshot shows a C++ development environment with the following details:

- Title Bar:** ND213 C03 L01 04.2 Cache Memory SC
- Code Editor:** A snippet of C++ code demonstrating memory access patterns. It includes a nested loop that fills a 4x4 matrix with values from 0 to 15 and prints the execution time.
- Output Terminal:** Shows the command "Execution time: 144 nanoseconds" and the prompt "bash-3.2\$".
- IDE Features:** Call Stack, Watch, Problems, Output, Debug Console, Terminal, Breakpoints, and various status icons at the bottom.

**Code Snippet:**

```
ND213 C03 L01 04.2 Cache Memory SC
example.cpp x
example.cpp > ...
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 = std::chrono::high_resolution_clock::now();
11    for (int i = 0; i < size; i++)
12    {
13        for (int j = 0; j < size; j++)
14        {
15            x[j][i] = i + j;
16            //std::cout << &x[j][i] << ":"; i=" << i << ", j=" << j << std::endl;
17        }
18    }
19
20    // print execution time to console
21    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
22    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(t2 - t1).count();
23    std::cout << "Execution time: " << duration << " nanoseconds" << std::endl;
24
25    return 0;
26 }
```

**Output Terminal:**

```
Execution time: 144 nanoseconds
bash-3.2$
```

**Bottom Status Bar:**

- Breakpoints: 1
- Play/Pause: ▶
- Volume: 🔊
- Time: 4:44 / 5:36
- C++: on catch
- CC:
- HD:
- YouTube:
- RSS:
- Settings:

Now, we see that it took the program 144 nanoseconds to execute this array here.

# Cache-friendly coding

In the code sample to the right, run the code and note the results. Then please modify the code slightly by interchanging the index `i` and `j` when accessing the variable `x` and take a close look at the resulting runtime performance compared to the original version.

## main.cpp

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 =
11        std::chrono::high_resolution_clock::now();
12    for (int i = 0; i < size; i++)
13    {
14        for (int j = 0; j < size; j++)
15        {
16            x[i][j] = i + j;
17            std::cout << &x[i][j] << ": i=" << i << ", "
18            j=" << j << std::endl;
19        }
20    }
21    // print execution time to console
22    auto t2 =
23        std::chrono::high_resolution_clock::now(); // stop
24        time measurement
```

Depending on the machine used for executing the two code versions, there will be a huge difference in execution time. In order to understand why this happens, let us revisit the memory layout we investigated with the gdb debugger at the beginning of this lesson: Even though we have created a two-dimensional array, it is stored in a one-dimensional succession of memory cells. In our minds, the array will (probably) look like this:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

## main.cpp

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 =
11        std::chrono::high_resolution_clock::now();
12    for (int i = 0; i < size; i++)
13    {
14        for (int j = 0; j < size; j++)
15        {
16            x[i][j] = i + j;
17            std::cout << &x[i][j] << ": i=" << i << ", "
18            j=" << j << std::endl;
19        }
20    }
21    // print execution time to console
22    auto t2 =
23        std::chrono::high_resolution_clock::now(); // stop
24        time measurement
25    auto duration =
```

\$ root@38b26905ca65:/home/

root@38b26905ca65:/home/workspace#

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

In memory however, it is stored as a single line as follows:

0,0		0,1		0,2		0,3		1,0		1,1		1,2		1,3		2,0		2,1		2,2
-----	--	-----	--	-----	--	-----	--	-----	--	-----	--	-----	--	-----	--	-----	--	-----	--	-----

As can be seen, the rows of the two-dimensional matrix are copied one after the other. This format is called "row major" and is the default for both C and C++. Some other languages such as Fortran are "column major" and a memory-aware programmer should always know the memory layout of the language he or she is

```

1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 =
11        std::chrono::high_resolution_clock::now();
12    for (int i = 0; i < size; i++)
13    {
14        for (int j = 0; j < size; j++)
15        {
16            x[i][j] = i + j;
17            std::cout << &x[i][j] << ": i=" << i << ", "
18            j=" << j << std::endl;
19        }
20    }
21    // print execution time to console
22    auto t2 =
23        std::chrono::high_resolution_clock::now(); // stop
24        time measurement
25    auto duration =

```

\$ root@38b26905ca65:/home/

root@38b26905ca65:/home/workspace#

Note that even though the row major memory layout is used in C++, this doesn't mean that all C++ libraries have the same default; for example, the popular Eigen library used for matrix operations in C++ defaults to column major.

```
20     // print execution time to console
21     auto t2 =
22         std::chrono::high_resolution_clock::now(); // stop
23         time measurement
24     auto duration =
```

```
$ root@38b26905ca65: /home/
```

```
root@38b26905ca65:/home/workspace#
```

As we have created an array of integers, the difference between two adjacent memory cells will be `sizeof(int)`, which is 4 bytes. Let us verify this by changing the size of the array to 4x4 and by plotting both the address and the index numbers to the console. Be sure to revert the array access back to

`x[i][j] = i + j`. You can plot by uncommenting the printout line in the inner for loop:

```
0x6021e0: i=0, j=0
0x6021e4: i=0, j=1
0x6021e8: i=0, j=2
0x6021ec: i=0, j=3
```

## main.cpp

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 =
11        std::chrono::high_resolution_clock::now();
12    for (int i = 0; i < size; i++)
13    {
14        for (int j = 0; j < size; j++)
15        {
16            x[i][j] = i + j;
17            std::cout << &x[i][j] << ": i=" << i << ", "
18            j=" << j << std::endl;
19        }
20    }
21    // print execution time to console
22    auto t2 =
23        std::chrono::high_resolution_clock::now(); // stop
24        time measurement
25    auto duration =
```

\$ root@38b26905ca65:/home/

root@38b26905ca65:/home/workspace#

```
0x6021f0: i=1, j=0  
0x6021f4: i=1, j=1  
0x6021f8: i=1, j=2  
0x6021fc: i=1, j=3
```

```
0x602200: i=2, j=0  
0x602204: i=2, j=1  
0x602208: i=2, j=2  
0x60220c: i=2, j=3
```

```
0x602210: i=3, j=0  
0x602214: i=3, j=1  
0x602218: i=3, j=2  
0x60221c: i=3, j=3
```

```
Execution time: 83  
microseconds
```

Clearly, the difference between two inner loop cycles is at 4 as predicted.

```
11     for (int i = 0; i < size; i++)  
12     {  
13         for (int j = 0; j < size; j++)  
14         {  
15             x[i][j] = i + j;  
16             std::cout << &x[i][j] << ": i=" << i << ",  
17             j=" << j << std::endl;  
18         }  
19     }  
20     // print execution time to console  
21     auto t2 =  
22         std::chrono::high_resolution_clock::now(); // stop  
23         time measurement  
24     auto duration =
```

```
$ root@38b26905ca65:/home/
```

```
root@38b26905ca65:/home/workspace#
```

When we interchange the indices `i` and `j` when accessing the array as

```
x[j]
[i] = i + j;

std::cout << &x[j]
[i] << ": i=" << j
<< ", j=" << i <<
std::endl;
```

we get the following output:

```
0x6021e0: i=0, j=0
0x6021f0: i=1, j=0
0x602200: i=2, j=0
0x602210: i=3, j=0

0x6021e4: i=0, j=1
0x6021f4: i=1, j=1
0x602204: i=2, j=1
0x602214: i=3, j=1
```

### main.cpp

```
1 #include <chrono>
2 #include <iostream>
3
4 int main()
5 {
6     // create array
7     const int size = 4;
8     static int x[size][size];
9
10    auto t1 =
11        std::chrono::high_resolution_clock::now();
12    for (int i = 0; i < size; i++)
13    {
14        for (int j = 0; j < size; j++)
15        {
16            x[i][j] = i + j;
17            std::cout << &x[i][j] << ": i=" << i << ","
18            j=" << j << std::endl;
19        }
20    }
21    // print execution time to console
22    auto t2 =
23        std::chrono::high_resolution_clock::now(); // stop
24        time measurement
25    auto duration =
```

\$ root@38b26905ca65:/home/

root@38b26905ca65:/home/workspace#

```
0x6021e8: i=0, j=2  
0x6021f8: i=1, j=2  
0x602208: i=2, j=2  
0x602218: i=3, j=2
```

```
0x6021ec: i=0, j=3  
0x6021fc: i=1, j=3  
0x60220c: i=2, j=3  
0x60221c: i=3, j=3
```

```
Execution time:  
115 microseconds
```

As can be see, the difference between two rows is now 0x10, which is 16 in the decimal system. This means that with each access to the matrix, four memory cells are skipped and the principle of spatial locality is violated. As a result, the wrong data is loaded into the L1 cache, leading to cache misses.

```
3  
4 int main()  
5 {  
6     // create array  
7     const int size = 4;  
8     static int x[size][size];  
9  
10    auto t1 =  
11        std::chrono::high_resolution_clock::now();  
12        for (int i = 0; i < size; i++)  
13        {  
14            for (int j = 0; j < size; j++)  
15            {  
16                x[i][j] = i + j;  
17                std::cout << &x[i][j] << ": i=" << i << ",  
j=" << j << std::endl;  
18            }  
19        }  
20        // print execution time to console  
21        auto t2 =  
22            std::chrono::high_resolution_clock::now(); // stop  
time measurement  
22        auto duration =
```

```
$ root@38b26905ca65: /home/
```

```
root@38b26905ca65:/home/workspace#
```

and costly reload operations - hence the significantly increased execution time between the two code samples. The difference in execution time of both code samples shows that cache-aware programming can increase code performance significantly.

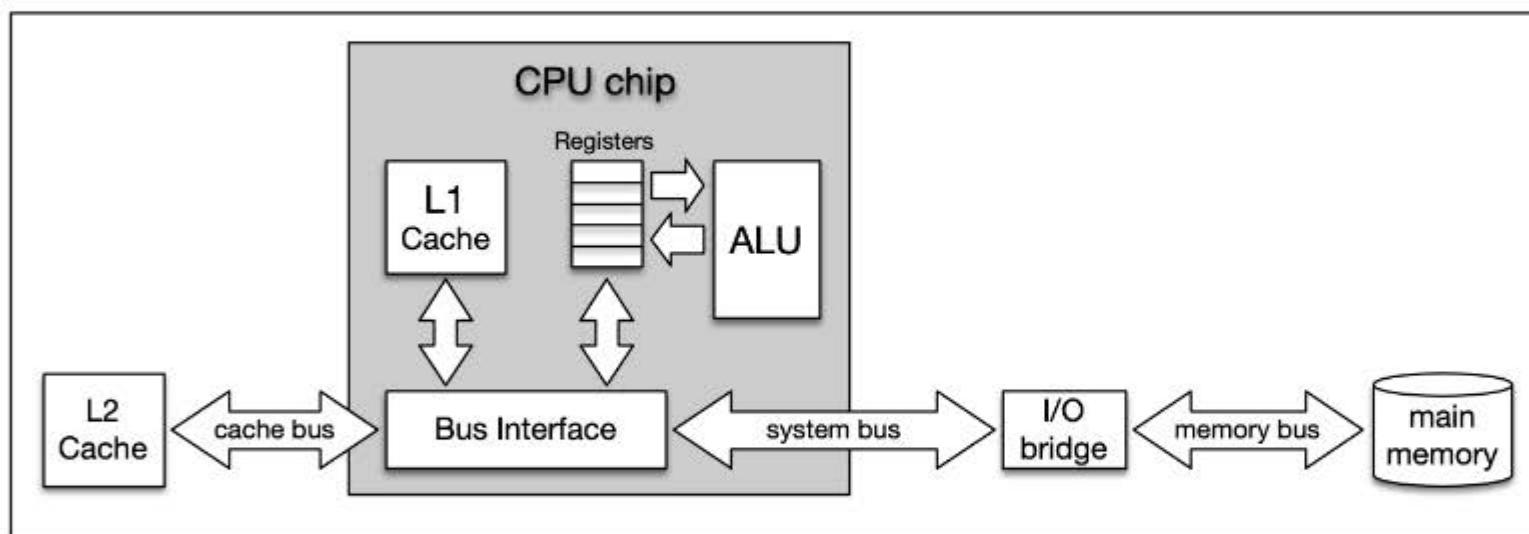
```
17      }
18  }
19
20  // print execution time to console
21  auto t2 =
    std::chrono::high_resolution_clock::now(); // stop
    time measurement
22  auto duration =
```

```
$ root@38b26905ca65: /home/|
```

```
root@38b26905ca65:/home/workspace#
```

## Cache Levels

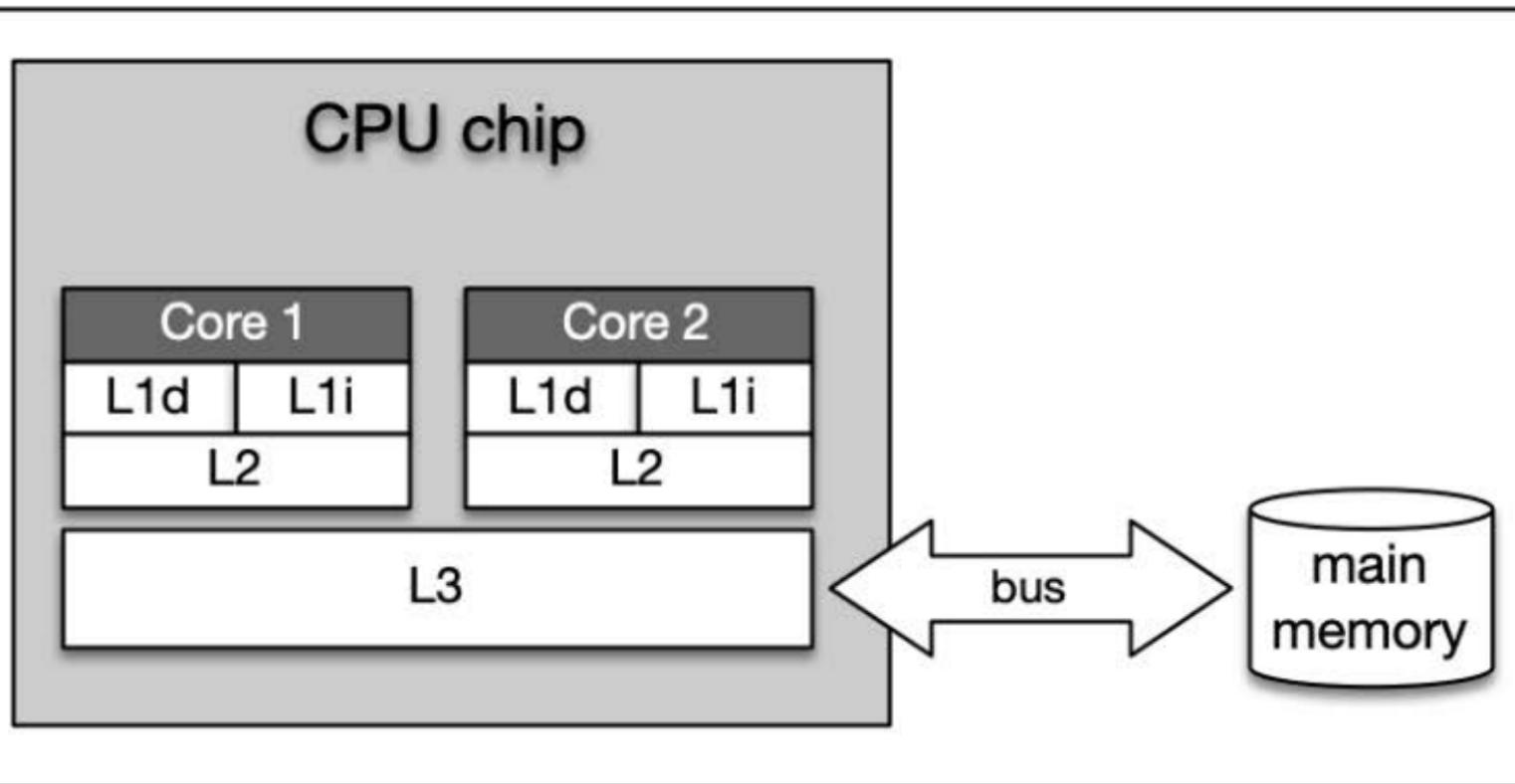
Cache memory is much faster but also significantly smaller than standard RAM. It holds the data that will (or might) be used by the CPU more often. In the memory hierarchy we have seen in the last section, the cache plays an intermediary role between fast CPU and slow RAM and hard disk. The figure below gives a rough overview of a typical system architecture:



System architecture diagram showing caches, ALU (arithmetic logic unit), main memory, and the buses connected each component.

The central CPU chip is connected to the outside world by a number of buses. There is a cache bus, which leads to a block denoted as L2 cache, and there is a system bus as well as a memory bus that leads to the computer main memory. The latter holds the comparatively large RAM while the L2 cache as well as the L1 cache are very small with the latter also being a part of the CPU itself.

The concept of L1 and L2 (and even L3) cache is further illustrated by the following figure, which shows a multi-core CPU and its interplay with L1, L2 and L3 caches:



L1, L2, and L3 cache

1. **Level 1 cache** is the fastest and smallest memory type in the cache hierarchy. In most systems, the L1 cache is not very large. Mostly it is in the range of 16 to 64 kBytes, where the memory areas for instructions and data are separated from each other (L1i and L1d, where "i" stands for "instruction" and "d" stands for "data". Also see "[Harvard architecture](#)" for further reference). The importance of the L1 cache grows with increasing speed of the CPU. In the L1 cache, the most frequently required instructions and data are buffered so that as few accesses as possible to the slow main memory are required. This cache avoids delays in data transmission and helps to make optimum use of the CPU's capacity.

2. **Level 2 cache** is located close to the CPU and has a direct connection to it. The information exchange between L2 cache and CPU is managed by the L2 controller on the computer main board. The size of the L2 cache is usually at or below 2 megabytes. On modern multi-core processors, the L2 cache is often located within the CPU itself. The choice between a processor with more clock speed or a larger L2 cache can be answered as follows: With a higher clock speed, individual programs run faster, especially those with high computing requirements. As soon as several programs run simultaneously, a larger cache is advantageous. Usually normal desktop computers with a processor that has a large cache are better served than with a processor that has a high clock rate.

3. **Level 3 cache** is shared among all cores of a multicore processor. With the L3 cache, the [cache coherence](#) protocol of multicore processors can work much faster. This protocol compares the caches of all cores to maintain data consistency so that all processors have access to the same data at the same time. The L3 cache therefore has less the function of a cache, but is intended to simplify and accelerate the cache coherence protocol and the data exchange between the cores.

On Mac, information about the system cache can be obtained by executing the command

`sysctl -a hw` in a terminal. On Debian Linux linux, this information can be found with

`lscpu | grep cache`. On my iMac Pro (2017), this command yielded (among others) the following output:

```
hw.memsize: 34359738368
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
hw.l2cachesize: 1048576
hw.l3cachesize: 14417920
```

- *hw.l1icachesize* is the size of the L1 instruction cache, which is at 32kB. This cache is strictly reserved for storing CPU instructions only.
- *hw.l1dcachesize* is also 32 KB and is dedicated for data as opposed to instructions.
- *hw.l2cachesize* and *hw.l3cachesize* show the size of the L2 and L3 cache, which are at 1MB and 14MB respectively.

It should be noted that the size of all caches combined is very small when compared to the size of the main memory (the RAM), which is at 32GB on my system.

Ideally, data needed by the CPU should be read from the various caches for more than 90% of all memory access operations. This way, the high latency of RAM and hard disk can be efficiently compensated.

## Temporal and Spatial Locality

The following table presents a rough overview of the latency of various memory access operations.

Even though these numbers will differ significantly between systems, the order of magnitude between the different memory types is noteworthy. While L1 access operations are close to the speed of a photon traveling at light speed for a distance of 1 foot, the latency of L2 access is roughly one order of magnitude slower already while access to main memory is two orders of magnitude slower.

0.5 ns	- CPU L1 dCACHE reference
1 ns	- speed-of-light (a photon) travel a 1 ft (30.5cm) distance
5 ns	- CPU L1 iCACHE Branch mispredict
7 ns	- CPU L2 CACHE reference
71 ns	- CPU cross-QPI/NUMA best case on XEON E5-46*
100 ns	- MUTEX lock/unlock
100 ns	- own DDR MEMORY reference
135 ns	- CPU cross-QPI/NUMA best case on XEON E7-*
202 ns	- CPU cross-QPI/NUMA worst case on XEON E7-*
325 ns	- CPU cross-QPI/NUMA worst case on XEON E5-46*
10,000 ns	- Compress 1K bytes with Zippy PROCESS
20,000 ns	- Send 2K bytes over 1 Gbps NETWORK
250,000 ns	- Read 1 MB sequentially from MEMORY
500,000 ns	- Round trip within a same DataCenter
10,000,000 ns	- DISK seek
10,000,000 ns	- Read 1 MB sequentially from NETWORK
30,000,000 ns	- Read 1 MB sequentially from DISK
150,000,000 ns	- Send a NETWORK packet CA → Netherlands

| | | ns |  
| | | us |  
| ms |

In algorithm design, programmers can exploit two principles to increase runtime performance:

1. **Temporal locality** means that address ranges that are accessed are likely to be used again in the near future. In the course of time, the same memory address is accessed relatively frequently (e.g. in a loop). This property can be used at all levels of the memory hierarchy to keep memory areas accessible as quickly as possible.
2. **Spatial locality** means that after an access to an address range, the next access to an address in the immediate vicinity is highly probable (e.g. in arrays). In the course of time, memory addresses that are very close to each other are accessed again multiple times. This can be exploited by moving the adjacent address areas upwards into the next hierarchy level during a memory access.

## Problems with physical memory

Virtual memory is a very useful concept in computer architecture because it helps with making your software work well given the configuration of the respective hardware on the computer it is running on.

The idea of virtual memory stems back from a (not so long ago) time, when the random access memory (RAM) of most computers was severely limited. Programmers needed to treat memory as a precious resource and use it most efficiently. Also, they wanted to be able to run programs even if there was not enough RAM available. At the time of writing (August 2019), the amount of RAM is no longer a large concern for most computers and programs usually have enough memory available to them. But in some cases, for example when trying to do video editing or when running multiple large programs at the same time, the RAM memory can be exhausted. In such a case, the computer can slow down drastically.

There are several other memory-related problems, that programmers need to know about:

- 1. Holes in address space :** If several programs are started one after the other and then shortly afterwards some of these are terminated again, it must be ensured that the freed-up space in between the remaining programs does not remain unused. If memory becomes too fragmented, it might not be possible to allocate a large block of memory due to a large-enough free contiguous block not being available any more.
- 2. Programs writing over each other :** If several programs are allowed to access the same memory address, they will overwrite each others' data at this location. In some cases, this might even lead to one program reading sensitive information (e.g. bank account info) that was written by another program. This problem is of particular concern when writing concurrent programs which run several threads at the same time.

The basic idea of virtual memory is to separate the addresses a program may use from the addresses in physical computer memory. By using a mapping function, an access to (virtual) program memory can be redirected to a real address which is guaranteed to be protected from other programs.

In the following, you will see, how virtual memory solves the problems mentioned above and you will also learn about the concepts of memory pages, frames and mapping. A sound knowledge on virtual memory will help you understand the C++ memory model, which will be introduced in the next lesson of this course.

## Quiz

On a 32-bit machine, each program has its own 32-bit address space. When a program wants to access a memory location, it must specify a 32-bit address, which directs it to the byte stored at this location.

On a hardware level, this address is transported to the physical memory via a parallel bus with 32 cables, i.e. each cable can either have the information 'high voltage', and 'low voltage' (or '1' and '0').

### QUIZ QUESTION

How large is the address space on a 32-bit system? What is the upper limit for program memory in GB?

1 GB

2 GB

4 GB

8 GB

SUBMIT



Thanks for completing that!

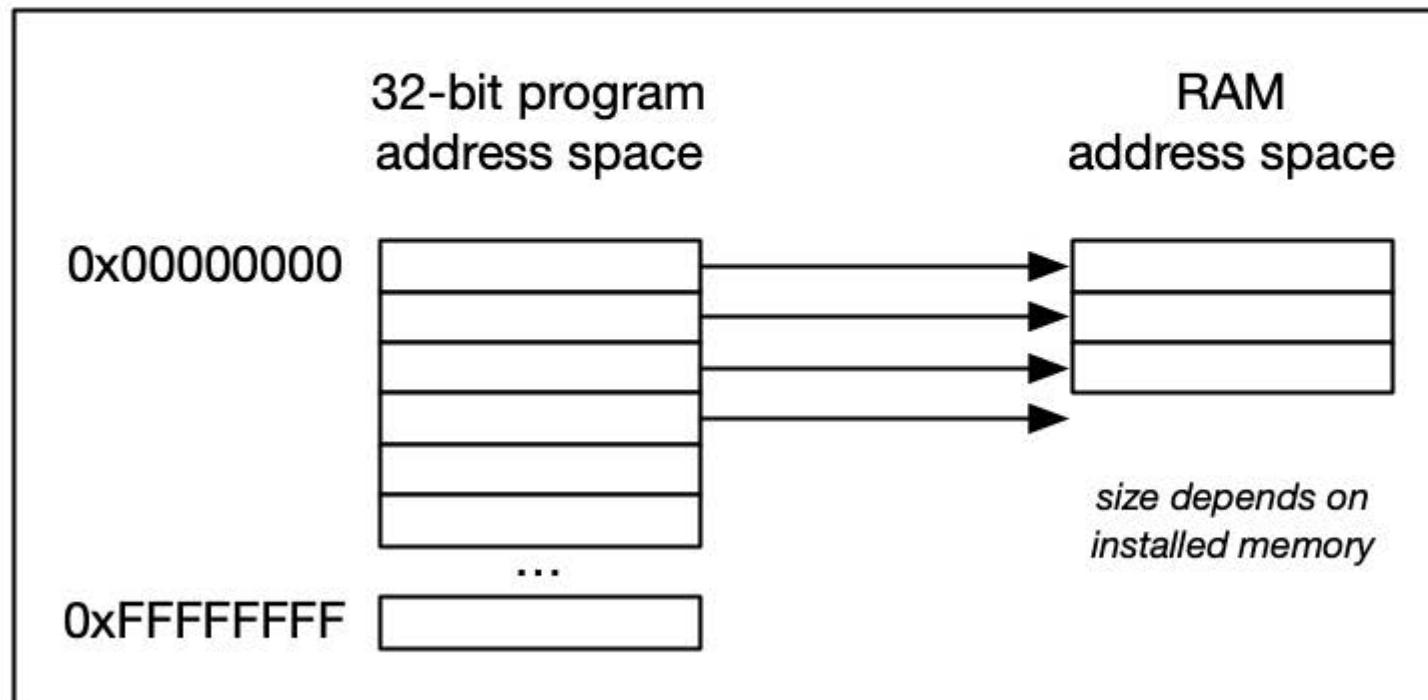
---

Correct!  $2^{32}$  bytes = 4GB; a 32-bit address space gives a program a (theoretical) total of 4 GB of memory it can address. In practice, the operating systems reserves some of this space however.

CONTINUE

## Expanding the available memory

As you have just learned in the quiz, the total amount of addressable memory is limited and depends on the architecture of the system (e.g. 32-bit). But what would happen if the available physical memory was below the upper bound imposed by the architecture? The following figure illustrates the problem for such a case:



In the image above, the available physical memory is less than the upper bound provided by the 32-bit address space.

On a typical architecture such as MIPS ("Microprocessor without interlocked pipeline stages"), each program is promised to have access to an address space ranging from 0x00000000 up to 0xFFFFFFFF. If however, the available physical memory is only 1GB in size, a 1-on-1 mapping would lead to undefined behavior as soon as the 30-bit RAM address space were exceeded.

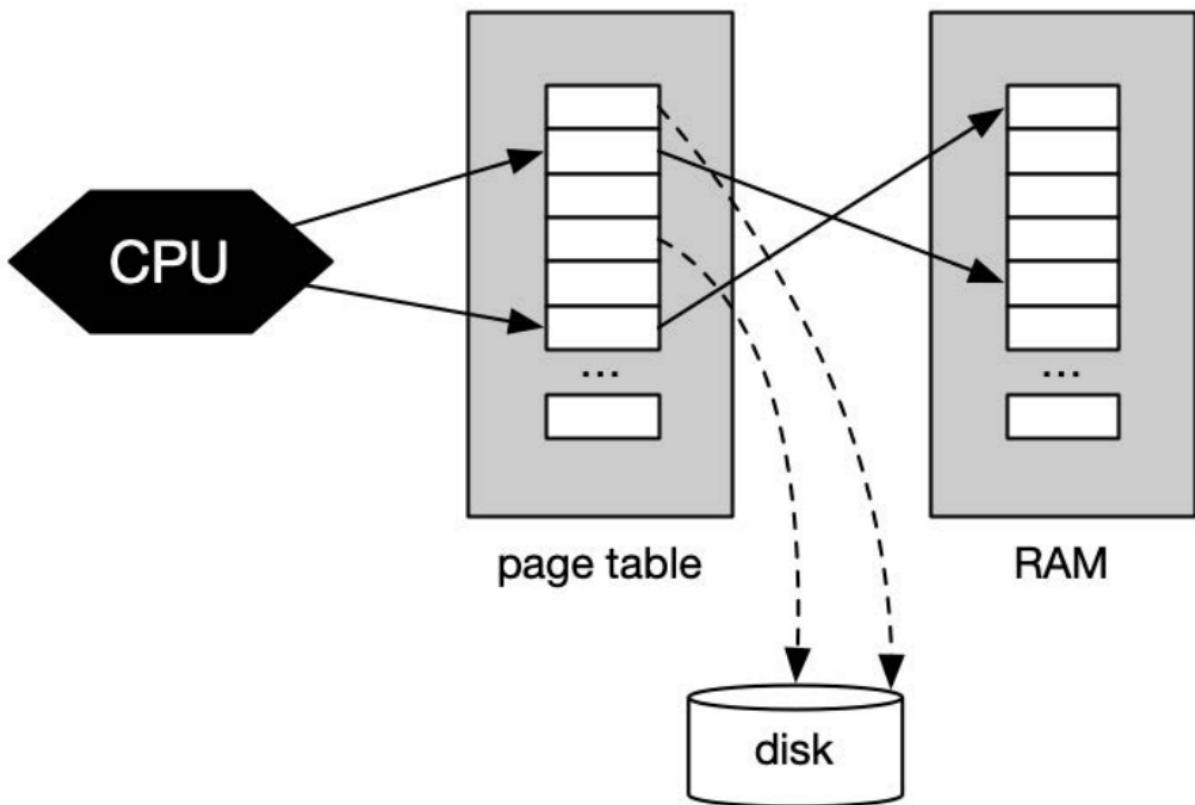
With virtual memory however, a mapping is performed between the virtual address space a program sees and the physical addresses of various storage devices such as the RAM but also the hard disk. Mapping makes it possible for the operating system to use physical memory for the parts of a process that are currently being used and back up the rest of the virtual memory to a secondary storage location such as the hard disk. With virtual memory, the size of RAM is not the limit anymore as the system hard disk can be used to store information as well.

The following figure illustrates the principle:

**virtual**  
address space

**physical**  
address space

32-bit program  
address space



With virtual memory, the RAM acts as a cache for the virtual memory space which resides on secondary storage devices. On Windows systems, the file `pagefile.sys` is such a virtual memory container of varying size. To speed up your system, it makes sense to adjust the system settings in a way that this file is stored on an SSD instead of a slow magnetic hard drive, thus reducing the latency. On a Mac, swap files are stored in `/private/var/vm/`.

In a nutshell, virtual memory guarantees us a fixed-size address space which is largely independent of the system configuration. Also, the OS guarantees that the virtual address spaces of different programs do not interfere with each other.

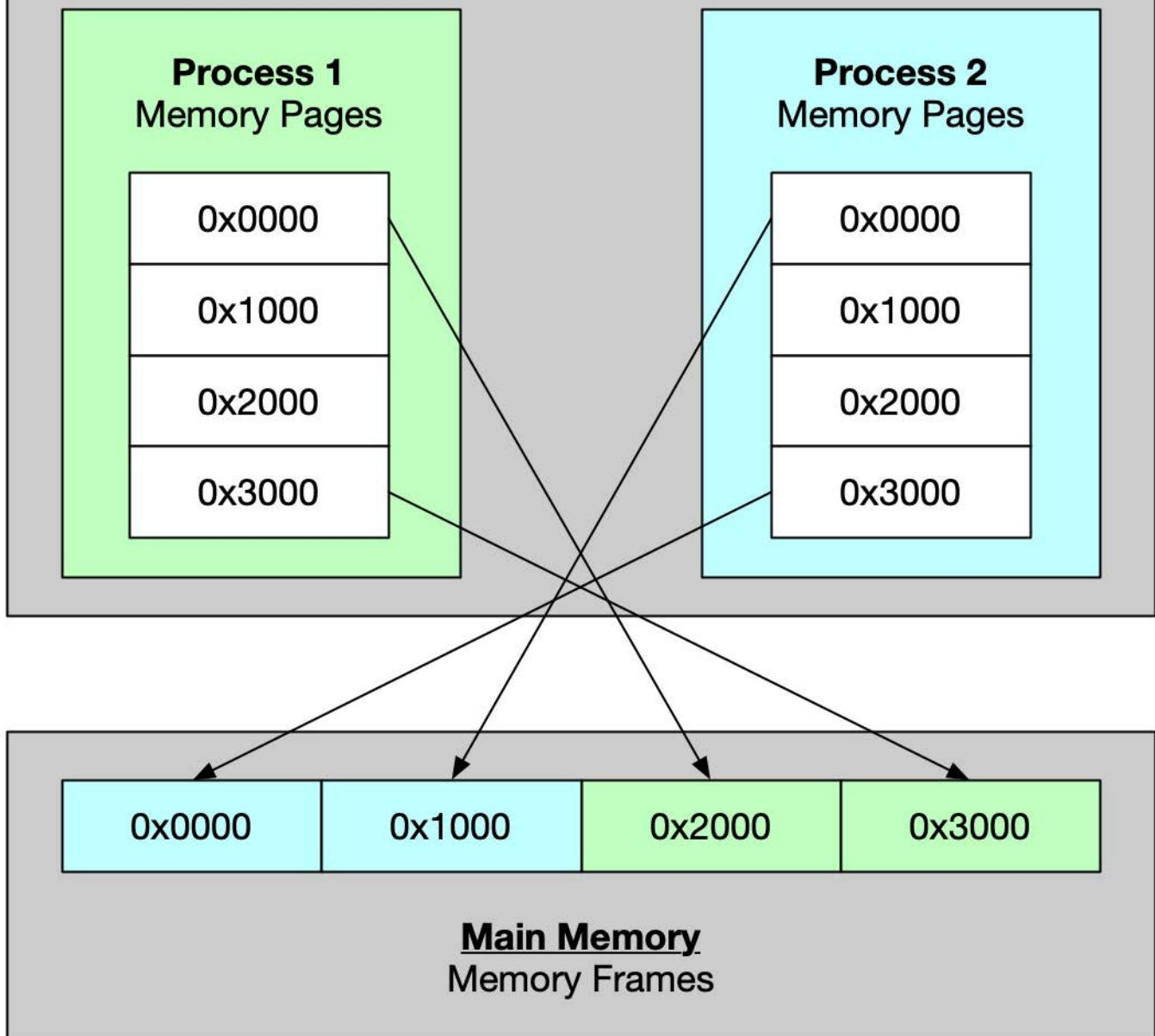
The task of mapping addresses and of providing each program with its own virtual address space is performed entirely by the operating system, so from a programmer's perspective, we usually don't have to bother much about memory that is being used by other processes.

Before we take a closer look at an example though, let us define two important terms which are often used in the context of caches and virtual memory:

- A **memory page** is a number of directly successive memory locations in virtual memory defined by the computer architecture and by the operating system. The computer memory is divided into memory pages of equal size. The use of memory pages enables the operating system to perform virtual memory management. The entire working memory is divided into tiles and each address in this computer architecture is interpreted by the Memory Management Unit (MMU) as a logical address and converted into a physical address.
- A **memory frame** is mostly identical to the concept of a memory page with the key difference being its location in the physical main memory instead of the virtual memory.

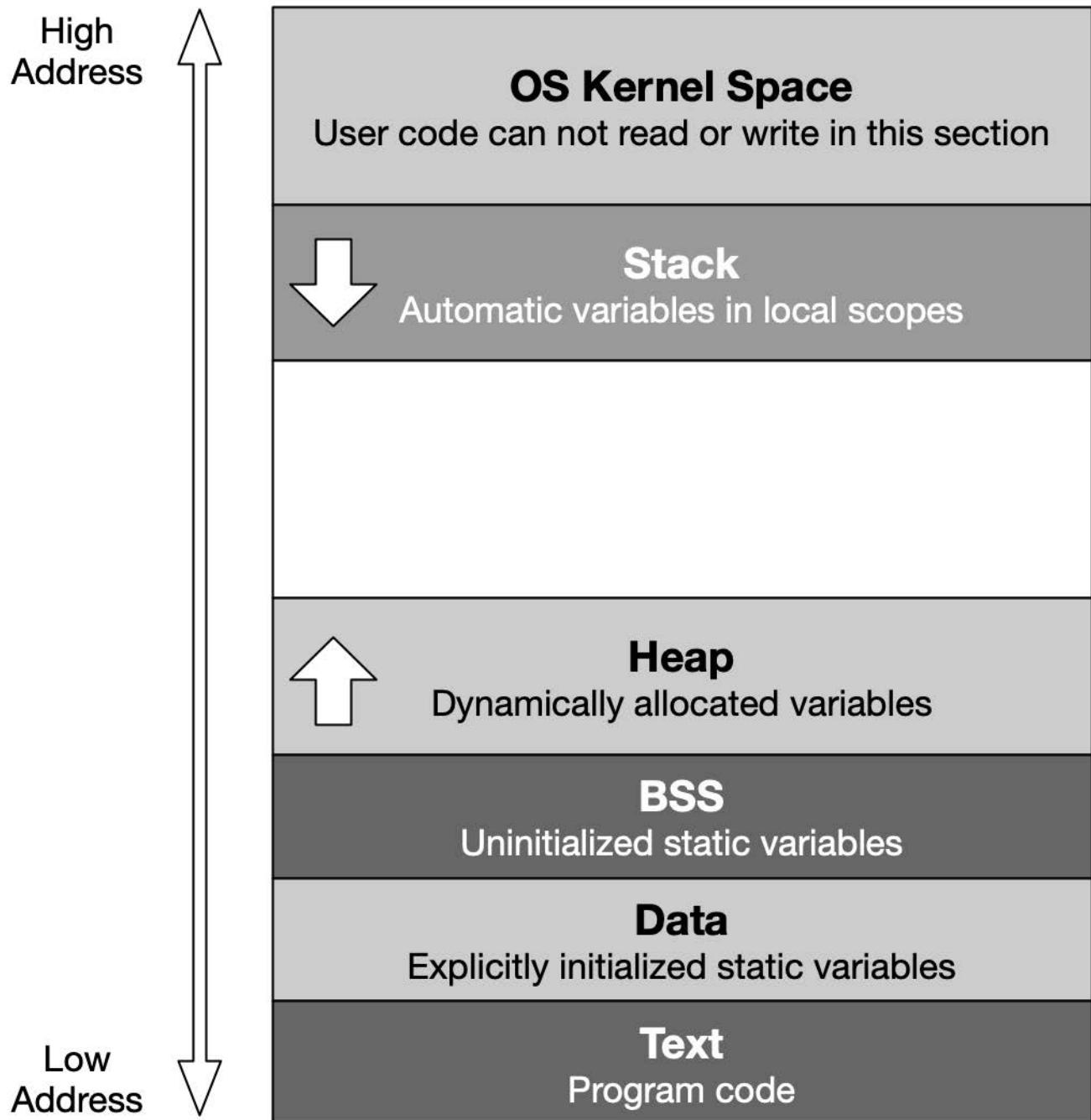
The following diagram shows two running processes and a collection of memory pages and frames:

## Virtual Memory



As can be seen, both processes have their own virtual memory space. Some of the pages are mapped to frames in the physical memory and some are not. If process 1 needs to use memory in the memory page that starts at address 0x1000, a page fault will occur if the required data is not there. The memory page will then be mapped to a vacant memory frame in physical memory. Also, note that the virtual memory addresses are not the same as the physical addresses. The first memory page of process 1, which starts at the virtual address 0x0000, is mapped to a memory frame that starts at the physical address 0x2000.

In summary, virtual memory management is performed by the operating system and programmers do usually not interfere with this process. The major benefit is a unique perspective on a chunk of memory for each program that is only limited in its size by the architecture of the system (32 bit, 64 bit) and by the available physical memory, including the hard disk.



## The Process Memory Model

As we have seen in the previous lesson, each program is assigned its own virtual memory by the operating system. This address space is arranged in a linear fashion with one block of data being stored at each address. It is also divided into several distinct areas as illustrated by the figure below:

The last address `0xFFFFFFFF` converts to the decimal `4.294.967.295`, which is the total amount of memory blocks that can theoretically addressed in a 32 bit operating system - hence the well-known limit of 4GB of memory. On a 64 bit system, the available space is significantly (!) larger. Also, the addresses are stored with 8 bytes instead of 4 bytes.

From a programming perspective though, we are not able to use the entire address space. Instead, the blocks "OS Kernel Space" and "Text" are reserved for the operating system. In kernel space, only the most trusted code is executed - it is fully maintained by the operating system and serves as an interface between the user code and the system kernel. In this course, we will not be directly concerned with this part of memory. The section called 'text' holds the program code generated by the compiler and linker. As with the kernel space, we will not be using this block directly in this course. Let us now take a look at the remaining blocks, starting from the top:

1. The **stack** is a contiguous memory block with a fixed maximum size. If a program exceeds this size, it will crash. The stack is used for storing automatically allocated variables such as local variables or function parameters. If there are multiple threads in a program, then each thread has its own stack memory. New memory on the stack is allocated when the path of execution enters a scope and freed again once the scope is left. It is important to know that the stack is managed "automatically" by the compiler, which means we do not have to concern ourselves with allocation and deallocation.
2. The **heap** (also called "free store" in C++) is where data with dynamic storage lives. It is shared among multiple threads in a program, which means that memory management for the heap needs to take concurrency into account. This makes memory allocations in the heap more complicated than stack allocations. In general, managing memory on the heap is more (computationally) expensive for the operating system, which makes it slower than stack memory. Contrary to the stack, the heap is not managed automatically by the system, but by the

programmer. If memory is allocated on the heap, it is the programmer's responsibility to free it again when it is no longer needed. If the programmer manages the heap poorly or not at all, there will be trouble.

3. The **BSS** (Block Started by Symbol) segment is used in many compilers and linkers for a segment that contains global and static variables that are initialized with zero values. This memory area is suitable, for example, for arrays that are not initialized with predefined values.
4. The **Data** segment serves the same purpose as the BSS segment with the major difference being that variables in the Data segment have been initialized with a value other than zero. Memory for variables in the Data segment (and in BSS) is allocated once when a program is run and persists throughout its lifetime.

## Memory Allocation in C++

Now that we have an understanding of the available process memory, let us take a look at memory allocation in C++.

Not every variable in a program has a permanently assigned area of memory. The term **allocate** refers to the process of assigning an area of memory to a variable to store its value. A variable is **deallocated** when the system reclaims the memory from the variable, so it no longer has an area to store its value.

Generally, three basic types of memory allocation are supported:

1. **Static memory allocation** is performed for static and global variables, which are stored in the BSS and Data segment. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
2. **Automatic memory allocation** is performed for function parameters as well as local variables, which are stored on the stack. Memory for these types of variables is allocated when the path of execution enters a scope and freed again once the scope is left.
3. **Dynamic memory allocation** is a possibility for programs to request memory from the operating system at runtime when needed. This is the major difference to automatic and static allocation, where the size of the variable must be known at compile time. Dynamic memory allocation is not performed on the limited stack but on the heap and is thus (almost) only limited by the size of the address space.

From a programmer's perspective, stack and heap are the most important areas of program memory. Hence, in the following lessons, let us look at these two in turn.

The screenshot shows a C++ development environment with the following details:

- Title Bar:** ND213 C03 L02 02.2 Automatic Memory Allocation (The Stack) SC
- File:** example.cpp
- Code Area:** Displays the following C++ code:

```
1 #include <stdio.h>
2
3 void MyFunc()
4 {
5     int k = 3;
6     printf ("3: %p \n",&k);
7 }
8
9 int main()
10 {
11     int i = 1;
12     printf ("1: %p \n",&i);
13
14     int j = 2;
15     printf ("2: %p \n",&j);
16
17     MyFunc();
18
19     int l = 4;
20     printf ("4: %p \n",&l);
21
22     return 0;
23 }
```
- Sidebar:** CALL STACK (selected), VARIABLES, SEARCH, FOLDERS, and FILE.
- Bottom Navigation:** WATCH, PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and BUILD and LAUNCH MM.
- Output Area:** Shows memory addresses for variables 1 through 4:

```
1: 0x7fffeefbff658
2: 0x7fffeefbff654
3: 0x7fffeefbff62c
4: 0x7fffeefbff650
bash-3.2$
```
- Text Overlay:** stored on the stack and now let's get back into the example.
- Bottom Icons:** CC, HD, YouTube, and other media controls.

## Properties of Stack Memory

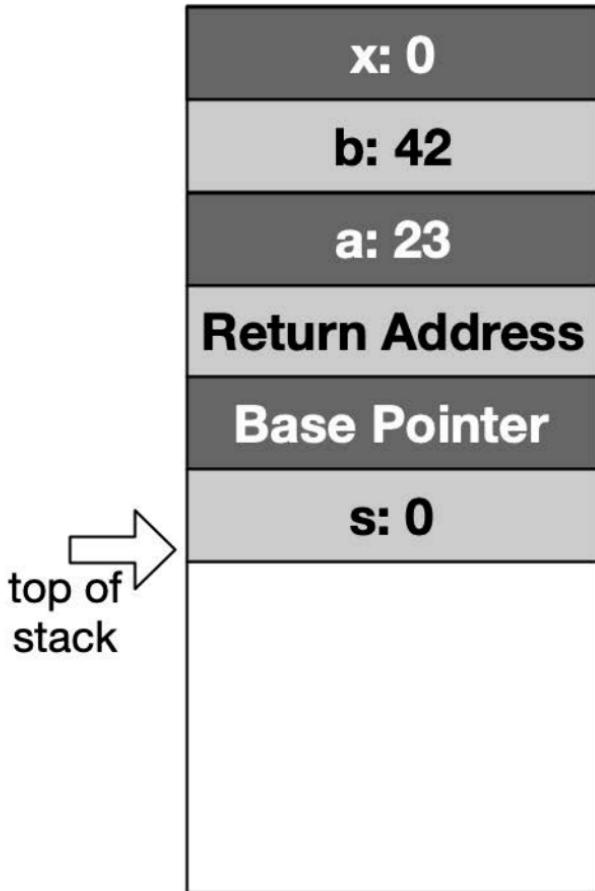
In the available literature on C++, the terms *stack* and *heap* are used regularly, even though this is not formally correct: C++ has the *free space*, *storage classes* and the *storage duration* of objects. However, since stack and heap are widely used in the C++ community, we will also use it throughout this course. Should you come across the above-mentioned terms in a book or tutorial on the subject, you now know that they refer to the same concepts as stack and heap do.

As mentioned in the last section, the stack is the place in virtual memory where the local variables reside, including arguments to functions. Each time a function is called, the stack grows (from top to bottom) and each time a function returns, the stack contracts. When using multiple threads (as in concurrent programming), it is important to know that each thread has its own stack memory - which can be considered thread-safe.

In the following, a short list of key properties of the stack is listed:

1. The stack is a **contiguous block of memory**. It will not become fragmented (as opposed to the heap) and it has a fixed maximum size.
2. When the **maximum size of the stack** memory is exceeded, a program will crash.
3. Allocating and deallocating **memory is fast** on the stack. It only involves moving the stack pointer to a new position.

The following diagram shows the stack memory during a function call:



```
int Add(int a, int b)
{
    int s = 0;
    s = a + b;

    return s;
}

int main()
{
    int x = 0;

    x = Add(23,42);

    return 0;
}
```

In the example, the variable `x` is created on the stack within the scope of `main`. Then, a stack frame which represents the function `Add` and its variables is pushed to the stack, moving the stack pointer further downwards. It can be seen that this includes the local variables `a` and `b`, as well as the return address, a base pointer and finally the return value `s`.



## Stack Growth and Contraction

In the first experiment, we will look at the behavior of the stack when local variables are allocated and a function is called. Consider the piece of code on the right.

Within the main function, we see two declarations of local variables `i` and `j` followed by a call to `MyFunc`, where another local variable is allocated. After `MyFunc` returns, another local variable is allocated in `main`. The program generates the following output:

```
1: 0x7ffeffbfff688
2: 0x7ffeffbfff684
3: 0x7ffeffbfff65c
4: 0x7ffeffbfff680
```



Between 1 and 2, the stack address is reduced by 4 bytes, which corresponds to the allocation of memory for the int `j`.

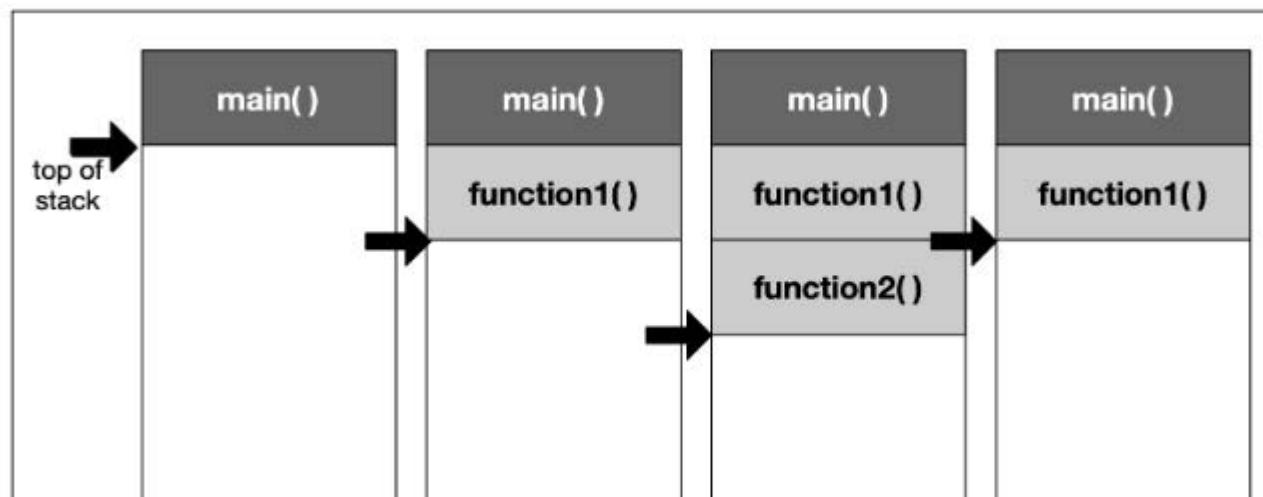
Between 2 and 3, the address pointer is moved by 0x28. We can easily see that calling a function causes a significant amount of memory to be allocated. In addition to the local variable of `MyFunc`, the computer needs to store additional data such as the

local variable of `main`, the return address, and the stack frame.

local variable of `MyFunc`, the compiler needs to store additional data such as the return address.

Between 3 and 4, `MyFunc` has returned and a third local variable `k` has been allocated on the stack. The stack pointer now has moved back to a location which is 4 bytes relative to position 2. This means that after returning from `MyFunc`, the stack has contracted to the size it had before the function call.

The following diagram illustrates how the stack grows and contracts during program execution:



## Total Stack Size

When a thread is created, stack memory is allocated by the operating system as a contiguous block. With each new function call or local variable allocation, the stack pointer is moved until eventually it will reach the bottom of said memory block. Once it exceeds this limit (which is called "stack overflow"), the program will crash. We will try to find out the limit of your computer's stack memory in the following exercise.

## Exercise: Create a Stack Overflow

Your task is to create a small program that allocates so much stack memory that an overflow happens. To do this, use a function that allocates some local variable and calls itself recursively. With each new function call, the address of the local variable shall be printed to the console along with the address of a local variable in main which has been allocated before the first function call.

The output of the program should look like this:

```
...
262011: stack bottom : 0x7fffeefbff688, current :
0x7fffeef400704
262012: stack bottom : 0x7fffeefbff688, current :
0x7fffeef4006e4
262013: stack bottom : 0x7fffeefbff688, current :
0x7fffeef4006c4
262014: stack bottom : 0x7fffeefbff688, current :
0x7fffeef4006a4
262015: stack bottom : 0x7fffeefbff688, current :
0x7fffeef400684
262016: stack bottom : 0x7fffeefbff688, current :
0x7fffeef400664
```

The left-most number keeps track of the recursion depth while the difference between the stack bottom and the current position of the stack pointer lets us compute the size of the stack memory which has been used up already. On my MacBook Pro, the size of the stack memory is at 8MB. On Mac or Linux systems, stack size can be checked using the command `ulimit -s` :

```
imac-pro:~ ahaja$ ulimit -s  
8192
```



On reaching the last line in the above output, the program crashed. As expected, the difference between stack bottom and current stack pointer corresponded to the maximum size of the stack:

```
0x7ffef400664 - 0x7ffefbff688 = 0xffffffff800FDC = 8.384.548  
bytes
```

From this experiment we can draw the simple conclusion that we do not want to run out of stack memory. This can happen quickly though, even on machines with large amounts of RAM installed. As we have seen, the size of the stack does not benefit from this at all but remains fixed at a very small size.



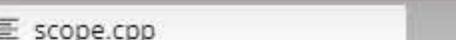
## Variable Scopes in C++

The time between allocation and deallocation is called the **lifetime** of a variable. Using a variable after its lifetime has ended is a common programming error, against which most modern languages try to protect: Local variables are only available within their respective scope (e.g. inside a function) and are simply not available outside - so using them inappropriately will result in a compile-time error. When using pointer variables however, programmers must make sure that allocation is handled correctly and that no invalid memory addresses are accessed.

The example to the right shows a set of local (or automatic) variables, whose lifetime is bound to the function they are in.

When `MyLocalFunction` is called, the local variable `isBelowThreshold` is allocated on the stack. When the function exits, it is again deallocated.

For the allocation of local variables, the following holds:



```
1 bool MyLocalFunction(int myInt)
2 {
3     bool isBelowThreshold = myInt < 42 ? true : false;
4     return isBelowThreshold;
5 }
6
7 int main()
8 {
9     bool res = MyLocalFunction(23);
10    return 0;
11 }
```

```
root@03bc9beecb7f:/home/v
```

```
root@03bc9beecb7f:/home/workspace#
```

1. Memory is allocated for local variables only after a function has been called. The parameters of a function are also local variables and they are initialized with a value copied from the caller.
2. As long as the current thread of execution is within function A, memory for the local variables remains allocated. This even holds true in case another function B is called from within the current function A and the thread of execution moves into this nested function call. However, within function B, the local variables of function A are not known.
3. When the function exits, its locals are deallocated and there is now way to them afterwards - even if the address were still known (e.g. by storing it within a pointer).

Let us briefly revisit the most common ways of passing parameters to a function, which are called *pass-by-reference* and *pass-by-value*.

## Quiz : How many local variables?

How many local variables are created within the scope of MyLocalFunction?

# Quiz : How many local variables?

How many local variables are created within the scope of MyLocalFunction?

**HIDE SOLUTION**

Two variables are created, namely myInt and isBelowThreshold .

## Passing Variables by Value

When calling a function as in the previous code example, its parameters (in this case `myInt`) are used to create local copies of the information provided by the caller. The caller is not sharing the parameter with the function but instead a proprietary copy is created using the assignment operator `=` (more about that later). When passing parameters in such a way, it is ensured that changes made to the local copy will not affect the original on the caller side. The upside to this is that inner workings of the function and the data owned by the caller are kept neatly separate.

However, there are two major downsides to this:

1. Passing parameters by value means that a copy is created, which is an expensive operation that might consume large amounts of memory, depending on the data that is being transferred. Later in this course we will encounter "move semantics", which is an effective way to compensate for this downside.
2. Passing by value also means that the created copy can not be used as a back channel for communicating with the caller, for example by directly writing the desired information into the variable.

Consider the example on the right in the `pass_by_value.cpp` file. In `main`, the integer `val` is initialized with 0. When passing it to the function `AddTwo`, a local copy of `val` is created, which only exists within the scope of `AddTwo`, so the addition operation has no effect on `val` on the caller side. So when `main` returns, `val` has a value of 2 instead of 4.

### pass\_by\_value.cpp

```
1 #include <iostream>
2
3 void AddTwo(int val)
4 {
5     val += 2;
6 }
7
8 int main()
9 {
10    int val = 0;
11    AddTwo(val);
12    val += 2;
13
14    std::cout << "val = " << val << std::endl;
15
16    return 0;
17 }
```

```
$ root@03bc9beecb7f:/home/v
```

```
root@03bc9beecb7f:/home/works$
```

However, with a slight modification, we can easily create a backchannel to the caller side. Consider the code on the right.

In this case, when passing the parameter to the function `AddThree`, we are creating a local copy as well but note that we are now passing a pointer variable. This means that a copy of the memory address of `val` is created, which we can then use to directly modify its content by using the dereference operator `*`.

## pass\_by\_pointer.cpp

```
1 #include <iostream>
2
3 void AddThree(int *val)
4 {
5     *val += 3;
6 }
7
8 int main()
9 {
10    int val = 0;
11    AddThree(&val);
12    val += 2;
13
14    std::cout << "val = " << val << std::endl;
15
16    return 0;
17 }
```

```
root@03bc9beecb7f:/home/v
```

```
root@03bc9beecb7f:/home/works#
```

## Passing Variables by Reference

The second major way of passing parameters to a function is by reference. With this way, the function receives a reference to the parameter, rather than a copy of its value. As with the example of `AddThree` above, the function can now modify the argument such that the changes also happen on the caller side. In addition to the possibility to directly exchange information between function and caller, passing variables by reference is also faster as no information needs to be copied, as well as more memory-efficient.

A major disadvantage is that the caller does not always know what will happen to the data it passes to a function (especially when the function code can not be modified easily). Thus, in some cases, special steps must be taken to protect ones data from inappropriate modification.

Let us now look at an example of passing a variable by reference, shown in the code on the right.

pass\_by\_reference.cpp

```
1 #include <iostream>
2
3 void AddFour(int &val)
4 {
5     val += 4;
6 }
7
8 int main()
9 {
10    int val = 0;
11    AddFour(val);
12    val += 2;
13
14    std::cout << "val = " << val << std::endl;
15
16    return 0;
17 }
```

root@03bc9beecb7f:/home/v

root@03bc9beecb7f:/home/works#

To pass a variable by reference, we simply declare the function parameters as references using `&` rather than as normal variables. When the function is called, `val` will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument!

```
1 #include <iostream>
2
3 void AddFour(int &val)
4 {
5     val += 4;
6 }
7
8 int main()
9 {
10    int val = 0;
11    AddFour(val);
12    val += 2;
13
14    std::cout << "val = " << val << std::endl;
15
16    return 0;
17 }
```

```
root@03bc9beecb7f:/home/v
```

```
root@03bc9beecb7f:/home/works#
```

## Quiz : Modifying several parameters

An additional advantage of passing variables by reference is the possibility to modify several variables. When using the function return value for such a purpose, returning several variables is usually very cumbersome.

Your task here is to create a function `AddFive` that modifies the `int` input variable by adding 5 and modifies the `bool` input variable to be `true`. In the code to the right you will find the function call in `main()`.

HIDE SOLUTION

```
void AddFive(int &val, bool &success)
{
    val += 5;
    success = true;
}
```

quiz.cpp

```
1 #include <iostream>
2
3 void AddFive(int &v, bool &s){
4     v = v + 5;
5     s = true;
6 }
7
8 int main()
9 {
10     int val = 0;
11     bool success = false;
12     AddFive(val, success);
13     val += 2;
14
15     std::cout << "val = " << val << ", success = " << success << std::endl;
16
17     return 0;
18 }
```

\$ root@03bc9beecb7f: /home/v

root@03bc9beecb7f:/home/works\$

## Pointers vs. References

As we have seen in the examples above, the use of pointers and references to directly manipulate function arguments in a memory-effective way is very similar. Let us compare the two methods in the code on the right.

As can be seen, pointer and reference are both implemented by using a memory address. In the case of `AddFour` the caller does not even realize that `val` might be modified while with `AddSix`, the reference to `val` has to be explicitly written by using `&`.

If passing by value needs to be avoided, both pointers and references are a way to achieve this. The following selection of properties contrasts the two methods so it will be easier to decide which to use from the perspective of the use-case at hand:

- Pointers can be declared without initialization. This means we can pass an uninitialized pointer to a function who then internally performs the initialization for us.
- Pointers can be reassigned to another memory block on the heap.
- References are usually easier to use (depending on the expertise level of the

```
1 #include <iostream>
2
3 void AddFour(int &val)
4 {
5     val += 4;
6 }
7
8 void AddSix(int *val)
9 {
10    *val += 6;
11 }
12
13 int main()
14 {
15     int val = 0;
16     AddFour(val);
17     AddSix(&val);
18
19     std::cout << "val = " << val << std::endl;
20
21     return 0;
22 }
```

```
root@03bc9beecb7f:/home/v
```

```
root@03bc9beecb7f:/home/works]
```

- References are usually easier to use (depending on the expertise level of the programmer). Sometimes however, if a third-party function is used without properly looking at the parameter definition, it might go unnoticed that a value has been modified.

Now, we will compare the three strategies we have seen so far with regard to stack memory usage. Consider the code on the right.

After creating a local variable `i` in `main` to give us the address of the stack bottom, we are passing `i` by-value to our first function. Inside `CallByValue`, the memory address of a local variable `j` is printed to the console, which serves as a marker for the stack pointer. With the second function call in `main`, we are passing a reference to `i` to `CallByPointer`. Lastly, the function `CallByReference` is called in `main`, which again takes the integer `i` as an argument. However, from looking at `main` alone, we can not tell whether `i` will be passed by value or by reference.

On my machine, when compiled with g++ (Apple clang version 11.0.0), the program produces the following output:

```
stack bottom: 0x7fffeefbf698
call-by-value: 0x7fffeefbf678
call-by-pointer: 0x7fffeefbf674
call-by-reference: 0x7fffeefbf674
```

```
1 #include <stdio.h>
2
3 void CallByValue(int i)
4 {
5     int j = 1;
6     printf ("call-by-value: %p\n",&j);
7 }
8
9 void CallByPointer(int *i)
10 {
11     int j = 1;
12     printf ("call-by-pointer: %p\n",&j);
13 }
14
15 void CallByReference(int &i)
16 {
17     int j = 1;
18     printf ("call-by-reference: %p\n",&j);
19 }
20
21 int main()
22 {
```

```
root@03bc9beecb7f:/home/v
```

```
root@03bc9beecb7f:/home/workspace#
```

```
stack bottom: 0x7ffeefbff698  
call-by-value: 0x7ffeefbf678  
call-by-pointer: 0x7ffeefbf674  
call-by-reference: 0x7ffeefbf674
```

Depending on your system, the compiler you use and the compiler optimization techniques, you may not always see this result. In some cases

Let us take a look at the respective differences to the stack bottom in turn:

1. `CallByValue` requires 32 bytes of memory. As discussed before, this is reserved for e.g. the function return address and for the local variables within the function (including the copy of `i`).
2. `CallByPointer` on the other hand requires - perhaps surprisingly - 36 bytes of memory. Let us complete the examination before going into more details on this result.
3. `CallByReference` finally has the same memory requirements as `CallByPointer`.

### stack\_usage.cpp

```
20  
21 int main()  
22 {  
23     int i = 0;  
24     printf ("stack bottom: %p\n",&i);  
25  
26     CallByValue(i);  
27  
28     CallByPointer(&i);  
29  
30     CallByReference(i);  
31  
32     return 0;  
33 }
```

```
$ root@03bc9beecb7f:/home/v
```

```
root@03bc9beecb7f:/home/workspace#
```

## Quiz: Why does CallByValue require more memory?

In this section, we have argued at length that passing a parameter by reference avoids a costly copy and should - in many situations - be preferred over passing a parameter by value. Yet, in the experiment above, we have witnessed the exact opposite.

Can you explain why?

**HIDE SOLUTION**

Let us take a look at the size of the various parameter types using the `sizeof` command:

```
printf("size of int: %lu\n", sizeof(int));
printf("size of *int: %lu\n", sizeof(int *));
```

The output here is

```
size of int: 4
```

The output here is

```
size of int: 4  
size of *int: 8
```



Obviously, the size of the pointer variable is larger than the actual data type. As my machine has a 64 bit architecture, an address requires 8 byte.

As an experiment, you could use the `-m32` compiler flag to build a 32 bit version of the program. This yields the following output:

```
size of int: 4  
size of *int: 4
```

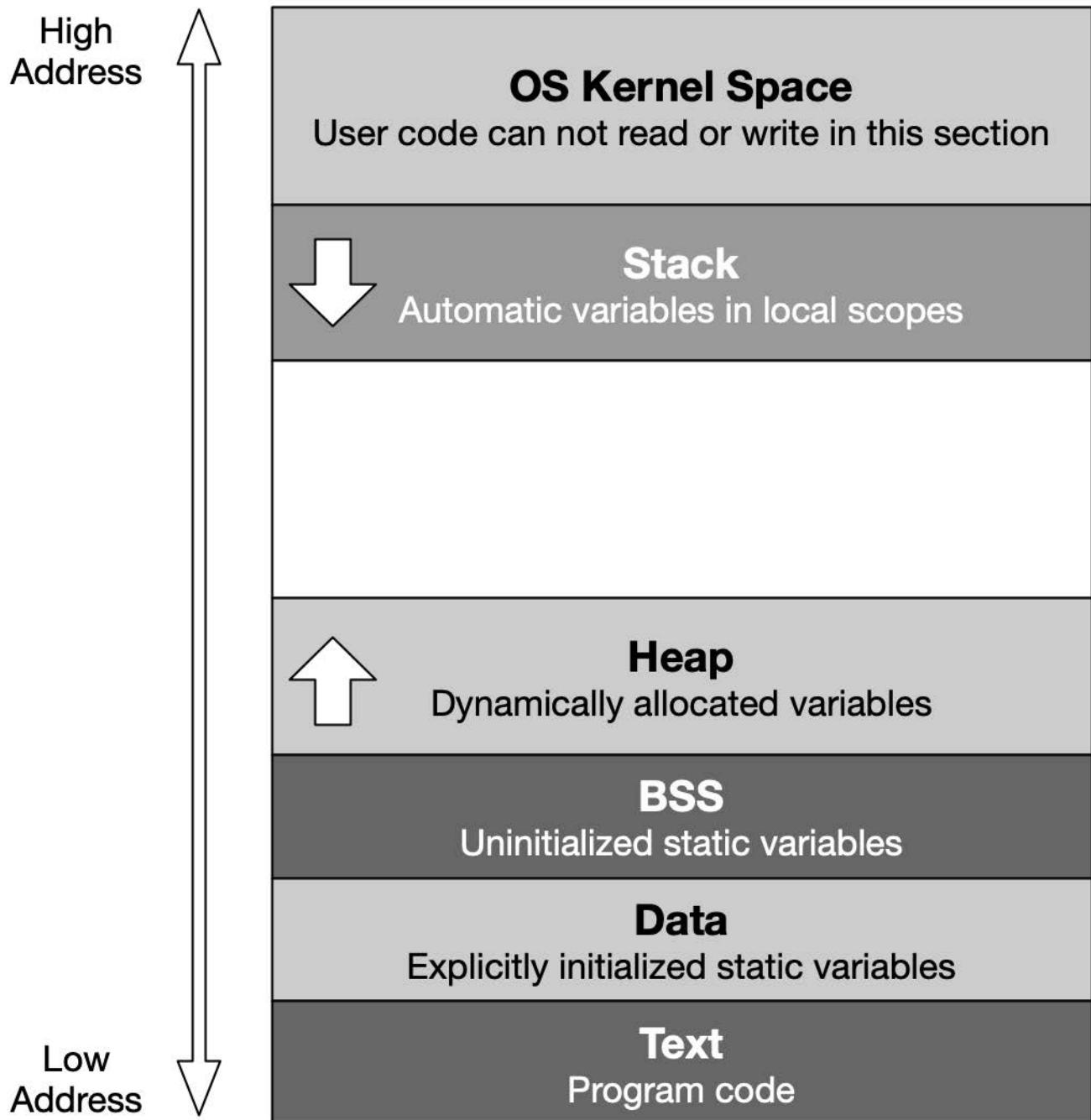


In order to benefit from call-by-reference, the size of the data type passed to the function has to surpass the size of the pointer on the respective architecture (i.e. 32 bit or 64 bit).

## Heap Memory

SEND FEEDBACK

Heap memory, also known as dynamic memory, is an important resource available to programs (and programmers) to store data. The following diagram again shows the layout of virtual memory with the heap being right above the BSS and Data segment.



As mentioned earlier, the heap memory grows upwards while the stack grows in the opposite direction. We have seen in the last lesson that the automatic stack memory shrinks and grows with each function call and local variable. As soon as the scope of a variable is left, it is automatically deallocated and the stack pointer is shifted upwards accordingly.

Heap memory is different in many ways: The programmer can request the allocation of memory by issuing a command such as `malloc` or `new` (more on that shortly). This block of memory will remain allocated until the programmer explicitly issues a command such as `free` or `delete`. The huge advantage of heap memory is the high degree of control a programmer can exert, albeit at the price of greater responsibility since memory on the heap must be actively managed.

Let us take a look at some properties of heap memory:

1. As opposed to local variables on the stack, memory can now be allocated in an arbitrary scope (e.g. inside a function) without it being deleted when the scope is left. Thus, as long as the address to an allocated block of memory is returned by a function, the caller can freely use it.
2. Local variables on the stack are allocated at compile-time. Thus, the size of e.g. a string variable might not be appropriate as the length of the string will not be known until the program is executed and the user inputs it. With local variables, a solution would be to allocate a long-enough array of and hope that the actual length does not exceed the buffer size. With dynamically allocated heap memory, variables are allocated at run-time. This means that the size of the above-mentioned string variable can be tailored to the actual length of the user input.

3. Heap memory is only constrained by the size of the address space and by the available memory.

With modern 64 bit operating systems and large RAM memory and hard disks the programmer commands a vast amount of memory. However, if the programmer forgets to deallocate a block of heap memory, it will remain unused until the program is terminated. This is called a "memory leak".

4. Unlike the stack, the heap is shared among multiple threads, which means that memory management for the heap needs to take concurrency into account as several threads might compete for the same memory resource.

5. When memory is allocated or deallocated on the stack, the stack pointer is simply shifted upwards or downwards. Due to the sequential structure of stack memory management, stack memory can be managed (by the operating system) easily and securely. With heap memory, allocation and deallocation can occur arbitrarily, depending on the lifetime of the variables. This can result in fragmented memory over time, which is much more difficult and expensive to manage.

## Memory Fragmentation

Let us construct a theoretic example of how memory on the heap can become fragmented: Suppose we are interleaving the allocation of two data types **X** and **Y** in the following fashion: First, we allocate a block of memory for a variable of type **X**, then another block for **Y** and so on in a repeated manner until some upper bound is reached. At the end of this operation, the heap might look like the following:

**blocks of memory**



At some point, we might then decide to deallocate all variables of type **Y**, leading to empty spaces in between the remaining variables of type **X**. In between two blocks of type "X", no memory for an additional "X" could now be squeezed in this example.

**blocks of memory**

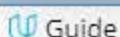


A classic symptom of memory fragmentation is that you try to allocate a large block and you can't, even though you appear to have enough memory free. On systems with virtual memory however, this is less of a problem, because large allocations only need to be contiguous in virtual address space, not in physical address space.

When memory is heavily fragmented however, memory allocations will likely take longer because the memory allocator has to do more work to find a suitable space for the new object.

Until now, our examples have been only theoretical. It is time to gain some practical experience in the next section using `malloc` and `free` as C-style methods for dynamic memory management.

So far we only considered primitive data types, whose storage space requirement was already fixed at compile time and could be scheduled with the building of the program executable. However, it is not always possible to plan the memory requirements exactly in advance, and it is inefficient to reserve the maximum memory space each time just to be on the safe side. C and C++ offer the option to reserve memory areas during the program execution, i.e. at runtime. It is important that the reserved memory areas are released again at the "appropriate point" to avoid memory leaks. It is one of the major challenges in memory management to always locate this "appropriate point" though.



Guide

## Allocating Dynamic Memory

To allocate dynamic memory on the heap means to make a contiguous memory area accessible to the program at runtime and to mark this memory as occupied so that no one else can write there by mistake.

To reserve memory on the heap, one of the two functions `malloc` (stands for *Memory Allocation*) or `calloc` (stands for *Cleared Memory Allocation*) is used. The header file `stdlib.h` or `malloc.h` must be included to use the functions.

Here is the syntax of `malloc` and `calloc` in C/C++:

```
pointer_name = (cast-type*) malloc(size);
pointer_name = (cast-type*) calloc(num_elems, size_elem);
```

`malloc` is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type `void` which can be cast into a pointer of any form.

`calloc` is used to dynamically allocate the specified number of blocks of memory of

the specified type. It initializes each block with a default value '0'.

Both functions return a pointer of type `void` which can be cast into a pointer of any form. If the space for the allocation is insufficient, a NULL pointer is returned.

In the code example on the right, a block of memory the size of an integer is allocated using `malloc`.

The `sizeof` command is a convenient way of specifying the amount of memory (in bytes) needed to store a certain data type. For an `int`, `sizeof` returns 4. However, when compiling this code, the following warning is generated on my machine:

```
warning: ISO C++ does not allow indirection on operand of type 'void *' [-Wvoid-ptr-dereference]
printf("address=%p, value=%d", p, *p);
```

In the virtual workspace, when compiling with `g++`, an error is thrown instead of a warning.

The problem with `void` pointers is that there is no way of knowing the offset to the end of the allocated memory block. For an `int`, this would be 4 bytes but for a `double`, the offset would be 8 bytes. So in order to retrieve the entire block of memory that has been reserved, we need to know the data type and the way to achieve this with `malloc` is by casting the return pointer:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p = (int*)malloc(sizeof(int));
7     //void *p = malloc(sizeof(int));
8     printf("address=%p, value=%d\n", p, *p);
9
10    // reserve memory for several integers
11    int *p2 = (int*)malloc(3*sizeof(int));
12    printf("address=%p, value=%d\n", p2, *p2);
13
14    int *p3 = (int*)malloc(3*sizeof(int));
15    p3[0] = 1; p3[1] = 2; p3[2] = 3;
16    printf("address=%p, second value=%d\n", p3, p3[1]);
17
18    return 0;
19 }
```

```
root@59a8c57c2a03:/home/v
```

```
root@59a8c57c2a03:/home/workspace#
```

```
int *p = (int*)malloc(sizeof(int));
```

This code now produces the following output without compiler warnings:

```
address=0x1003001f0, value=0
```

Obviously, the memory has been initialized with 0 in this case. However, you should not rely on pre-initialization as this depends on the data type as well as on the compiler you are using.

At compile time, only the space for the pointer is reserved (on the stack). When the pointer is initialized, a block of memory of `sizeof(int)` bytes is allocated (on the heap) at program runtime. The pointer on the stack then points to this memory location on the heap.

## Quiz

Modify the example in a way that memory for 3 integers is reserved.

HIDE SOLUTION

```
// reserve memory for several integers
int *p2 = (int*)malloc(3*sizeof(int));
printf("address=%p, value=%d\n", p2, *p2);
```

### malloc\_example.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int *p = (int*)malloc(sizeof(int));
7     //void *p = malloc(sizeof(int));
8     printf("address=%p, value=%d\n", p, *p);
9
10    // reserve memory for several integers
11    int *p2 = (int*)malloc(3*sizeof(int));
12    printf("address=%p, value=%d\n", p2, *p2);
13
14    int *p3 = (int*)malloc(3*sizeof(int));
15    p3[0] = 1; p3[1] = 2; p3[2] = 3;
16    printf("address=%p, second value=%d\n", p3, p3[1]);
17
18    return 0;
19 }
```

```
$ root@59a8c57c2a03: /home/v
```

```
root@59a8c57c2a03:/home/workspace# []
```

# Memory for Arrays and Structs

Since arrays and pointers are displayed and processed identically internally, individual blocks of data can also be accessed using array syntax:

```
int *p = (int*)malloc(3*sizeof(int));
p[0] = 1; p[1] = 2; p[2] = 3;
printf("address=%p, second value=%d\n", p, p[1]);
```

Until now, we have only allocated memory for a C/C++ data primitive (i.e. `int`). However, we can also define a proprietary structure which consists of several primitive data types and use `malloc` or `calloc` in the same manner as before:

```
struct MyStruct {
    int i;
    double d;
    char a[5];
};
```

```
MyStruct *p = (MyStruct*)calloc(4,sizeof(MyStruct));  
p[0].i = 1; p[0].d = 3.14159; p[0].a[0] = 'a';
```

After defining the struct `MyStruct` which contains a number of data primitives, a block of memory four times the size of `MyStruct` is created using the `calloc` command. As can be seen, the various data elements can be accessed very conveniently.

The size of the memory area reserved with `malloc` or `calloc` can be increased or decreased with the `realloc` function.

```
pointer_name = (cast-type*) realloc( (cast-type*)old_memblock, new_size );
```

To do this, the function must be given a pointer to the previous memory area and the new size in bytes. Depending on the compiler, the reserved memory area is either (a) expanded or reduced internally (if there is still enough free heap after the previously reserved memory area) or (b) a new memory area is reserved in the desired size and the old memory area is released afterwards.

The data from the old memory area is retained, i.e. if the new memory area is larger, the data will be available within new memory area as well. If the new memory area is smaller, the data from the old area will be available only up until the site of the new area - the rest is lost.

In the example on the right, a block of memory of initially 8 bytes (two integers) is resized to 16 bytes (four integers) using `realloc`.

Note that `realloc` has been used to increase the memory size and then

### realloc\_example.cpp

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     // reserve memory for two integers
7     int *p = (int*)malloc(2*sizeof(int));
8     p[0] = 1; p[1] = 2;
9
10    // resize memory to hold four integers
11    p = (int*)realloc(p,4*sizeof(int));
12    p[2] = 3; p[3] = 4;
13
14    // resize memory again to hold two integers
15    p = (int*)realloc(p,2*sizeof(int));
16
17    printf("address=%p, value=%d\n", p+0, *(p+0)); // valid
18    printf("address=%p, value=%d\n", p+1, *(p+1)); // valid
19
20    printf("address=%p, value=%d\n", p+2, *(p+2)); // INVALID
21    printf("address=%p, value=%d\n", p+3, *(p+3)); // INVALID
22}
```

```
$ root@59a8c57c2a03:/home/v
```

```
root@59a8c57c2a03:/home/workspace#
```

In the example on the right, a block of memory of initially 8 bytes (two integers) is resized to 16 bytes (four integers) using `realloc`.

Note that `realloc` has been used to increase the memory size and then decrease it immediately after assigning the values 3 and 4 to the new blocks. The output looks like the following:

```
address=0x100300060, value=1
address=0x100300064, value=2
address=0x100300068, value=3
address=0x10030006c, value=4
```

Interestingly, the pointers `p+2` and `p+3` can still access the memory location they point to. Also, the original data (numbers 3 and 4) is still there. So `realloc` will not erase memory but merely mark it as "available" for future allocations. It should be noted however that accessing a memory location *after* such an operation must be avoided as it could cause a segmentation fault. We will encounter segmentation faults soon when we discuss "dangling pointers" in one of the next lessons.

### realloc\_example.cpp

```
6 // dynamically allocate memory for two integers
7 int *p = (int*)malloc(2*sizeof(int));
8 p[0] = 1; p[1] = 2;
9
10 // resize memory to hold four integers
11 p = (int*)realloc(p,4*sizeof(int));
12 p[2] = 3; p[3] = 4;
13
14 // resize memory again to hold two integers
15 p = (int*)realloc(p,2*sizeof(int));
16
17 printf("address=%p, value=%d\n", p+0, *(p+0)); // valid
18 printf("address=%p, value=%d\n", p+1, *(p+1)); // valid
19
20 printf("address=%p, value=%d\n", p+2, *(p+2)); // INVALID
21 printf("address=%p, value=%d\n", p+3, *(p+3)); // INVALID
22
23 return 0;
24 }
```

```
root@59a8c57c2a03:/home/v
```

```
root@59a8c57c2a03:/home/workspace#
```

## Freeing up Memory

If memory has been reserved, it should also be released as soon as it is no longer needed. If memory is reserved regularly without releasing it again, the memory capacity may be exhausted at some point. If the RAM memory is completely used up, the data is swapped out to the hard disk, which slows down the computer significantly.

The `free` function releases the reserved memory area so that it can be used again or made available to other programs. To do this, the pointer pointing to the memory area to be freed is specified as a parameter for the function. In the `free_example.cpp`, a memory area is reserved and immediately released again.

```
free_example.cpp
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     void *p = malloc(100);
7     free(p);
8
9     return 0;
10 }
```

```
5. root@59a8c57c2a03: /home/v
```

```
root@59a8c57c2a03:/home/workspace#
```

Some things should be considered with dynamic memory management, whose neglect in some cases might result in unpredictable program behavior or a system crash - in some cases unfortunately without error messages from the compiler or the operating system:

1. `free` can only free memory that was reserved with `malloc` or `calloc`.
2. `free` can only release memory that has not been released before.  
Releasing the same block of memory twice will result in an error.

In the example on the right, a pointer `p` is copied into a new variable `p2`, which is then passed to `free` AFTER the original pointer has been already released.

```
free(41143,0x1000a55c0) malloc: *** error for object  
0x1003001f0: pointer being freed was not allocated.
```

In the workspace, you will see this error:

```
*** Error in './a.out': double free or corruption  
(fasttop): 0x000000000755010 ***
```

### free\_example.cpp

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     void *p = malloc(100);  
7     free(p);  
8  
9     return 0;  
10 }
```

```
root@59a8c57c2a03:/home/v
```

```
root@59a8c57c2a03:/home/workspace#
```

The pointer `p2` in the example is invalid as soon as `free(p)` is called. It still holds the address to the memory location which has been freed, but may not access it anymore. Such a pointer is called a "dangling pointer".

3. Memory allocated with `malloc` or `calloc` is not subject to the familiar rules of variables in their respective scopes. This means that they exist independently of block limits until they are released again or the program is terminated. However, the pointers which refer to such heap-allocated memory are created on the stack and thus only exist within a limited scope. As soon as the scope is left, the pointer variable will be lost - but not the heap memory it refers to.

## Quiz : Dynamic Memory Management with `malloc`, `calloc`, `resize` and `free`

Question 1: Match the code snippets to the respective comments

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

int main()
{
    // (X)
    int *m = (int*)malloc(sizeof(int));
    m = NULL;

    // (Y)
    int *n = (int*)malloc(sizeof(int));
    free(n);
    *n = 23;

    // (Z)
    char *o;
    *o = 'a';

    return 0;
}
```

Comments:

1. uses a dangling pointer
2. uses an uninitialized pointer
3. generates a memory leak

### QUESTION 1 OF 2

In the code above, there are three snippets marked with X, Y, and Z. Below the code there are three comments. Match the code snippets to the respective comments. Which pairing of comments and code snippets is the correct one?

X-1, Y-3, Z-2

X-2, Y-1, Z-3

X-3, Y-2, Z-1

X-3, Y-1, Z-2

SUBMIT

## Question 2 : Problems with pointers

```
int *f1(void)
{
    int x = 10;
    return (&x);
}

int *f2(void)
{
    int *px;
    *px = 10;
    return px;
}

int *f3(void)
{
    int *px;
    px = (int *)malloc(sizeof(int));
    *px = 10;
    return px;
}
```

QUESTION 2 OF 2

Which of the functions above is likely to cause pointer-related problems?

only `f3`

`f1` and `f3`

`f1` and `f2`

`f1`, `f2`, and `f3`

SUBMIT



Thanks for completing that!

Great work! In `f1`, the pointer variable `x` is a local variable to `f1`, and `f1` returns the pointer to that variable. `x` can disappear after `f1()` is returned if `x` exists on the stack. So `&x` can become invalid.

In `f2`, the pointer variable `px` is assigned a value without allocating its space.

`f3` works fine. Memory is allocated to the pointer variable `px` using `malloc()`. So, `px` exists on the heap, its existence will remain in memory even after the return of `f3()` as it is on the heap.

CONTINUE

Note that new and delete are operators while malloc and free



0:05 / 0:31



YouTube



are functions taken from C. We will compare in detail

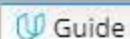


0:07 / 0:31



YouTube





Guide

## Comparing `malloc` with `new`

The functions `malloc` and `free` are library function and represent the default way of allocating and deallocating memory in C. In C++, they are also part of the standard and can be used to allocate blocks of memory on the heap.

With the introduction of classes and object oriented programming in C++ however, memory allocation and deallocation has become more complex: When an object is created, its constructor needs to be called to allow for member initialization. Also, on object deletion, the destructor is called to free resources and to allow for programmer-defined clean-up tasks. For this reason, C++ introduces the operators `new` / `delete`, which represent the object-oriented counterpart to memory management with `malloc` / `free`.

If we were to create a C++ object with `malloc`, the constructor and destructor of such an object would not be called. Consider the class on the right. The constructor allocates memory for the private element `_number` (yes, we could have simply used `int` instead of `int*`, but that's for educational purposes only), and the destructor releases memory again. The setter method `setNumber` finally assigns a value to `_number` under the assumption that memory has been allocated previously.

In main, we will allocate memory for an instance of `MyClass` using both `malloc` / `free` and `new` / `delete`.

With `malloc`, the program crashes on calling the method `setNumber`, as no memory has been allocated for `_number` - because the constructor has not been called. Hence, an `EXC_BAD_ACCESS` error occurs, when trying to access the memory location to which `_number` is pointing. With `_new`, the output looks like the following:

```
Allocate memory
Number: 42
Delete memory
```

```
1 #include <stdlib.h>
2 #include <iostream>
3
4 class MyClass
5 {
6     private:
7         int *_number;
8
9 public:
10    MyClass()
11    {
12        std::cout << "Allocate memory\n";
13        _number = (int *)malloc(sizeof(int));
14    }
15    ~MyClass()
16    {
17        std::cout << "Delete memory\n";
18        free(_number);
19    }
20    void setNumber(int number)
21    {
22        *_number = number;
23        std::cout << "Number: " << *_number << "\n";
24    }
25 };
```

```
root@efb13975c180:/home/workspace# rm overloading_array
root@efb13975c180:/home/workspace# []
```

### malloc\_v\_new.cpp

```
21     {
22         *_number = number;
23         std::cout << "Number: " << *_number << "\n";
24     }
25 };
26
27 int main()
28 {
29     // allocate memory using malloc
30     // comment these lines out to run the example below
31     //MyClass *myClass = (MyClass *)malloc(sizeof(MyClass));
32     //myClass->setNumber(42); // EXC_BAD_ACCESS
33     //free(myClass);
34
35     // allocate memory using new
36     MyClass *myClass = new MyClass();
37     myClass->setNumber(42); // works as expected
38     delete myClass;
39
40     return 0;
41 }
```

Before we go into further details of `new / delete`, let us briefly summarize the major differences between `malloc / free` and `new / delete`:

1. **Constructors / Destructors** Unlike `malloc( sizeof(MyClass) )`, the call `new MyClass()` calls the constructor. Similarly, `delete` calls the destructor.
2. **Type safety** `malloc` returns a void pointer, which needs to be cast into the appropriate data type it points to. This is not type safe, as you can freely vary the pointer type without any warnings or errors from the compiler as in the following small example:

```
MyObject *p = (MyObject*)malloc(sizeof(int));
```

In C++, the call `MyObject *p = new MyObject()` returns the correct type automatically - it is thus type-safe.

3. **Operator Overloading** As `malloc` and `free` are functions defined in a library, their behavior can not be changed easily. The `new` and `delete` operators however can be overloaded by a class in order to include optional proprietary behavior. We will look at an example of overloading `new` further down in this section.

# Creating and Deleting Objects

As with `malloc` and `free`, a call to `new` always has to be followed by a call to `delete` to ensure that memory is properly deallocated. If the programmer forgets to call `delete` on the object (which happens quite often, even with experienced programmers), the object resides in memory until the program terminates at some point in the future causing a *memory leak*.

Let us revisit a part of the code example to the right:

```
myClass = new MyClass();
myClass->setNumber(42); // works as expected
delete myClass;
```



The call to `new` has the following consequences:

1. Memory is allocated to hold a new object of type `MyClass`
2. A new object of type `MyClass` is constructed within the allocated memory by calling the constructor of `MyClass`

The call to `delete` causes the following:

1. The object of type `MyClass` is destroyed by calling its destructor
2. The memory which the object was placed in is deallocated

## Optimizing Performance with `placement new`

In some cases, it makes sense to separate memory allocation from object construction. Consider a case where we need to reconstruct an object several times. If we were to use the standard `new / delete` construct, memory would be allocated and freed unnecessarily as only the content of the memory block changes but not its size. By separating allocation from construction, we can get a significant performance increase.

C++ allows us to do this by using a construct called *placement new*: With `placement new`, we can pass a preallocated memory and construct an object at that memory location. Consider the following code:

```
void *memory = malloc(sizeof(MyClass));
MyClass *object = new (memory) MyClass;
```

The syntax `new (memory)` is denoted as *placement new*. The difference to the "conventional" `new` we have been using so far is that no memory is allocated. The call constructs an object and places it in the assigned memory location. There is however, no `delete` equivalent to `placement new`, so we have to call the destructor explicitly in this case instead of using `delete` as we would have done with a regular call to `new`:



```
object->~MyClass();  
free(memory);
```

Important: Note that this should never be done outside of `placement new`.

## Overloading new and delete

One of the major advantages of `new / delete` over `free / malloc` is the possibility of overloading. While both `malloc` and `free` are function calls and thus can not be changed easily, `new` and `delete` are operators and can thus be overloaded to integrate customized functionality, if needed.

The syntax for overloading the `new` operator looks as follows:

```
void* operator new(size_t size);
```

The operator receives a parameter `size` of type `size_t`, which specifies the number of bytes of memory to be allocated. The return type of the overloaded `new` is a void pointer, which references the beginning of the block of allocated memory.

The syntax for overloading the `delete` operator looks as follows:

```
void operator delete(void*);
```

The operator takes a pointer to the object which is to be deleted. As opposed to `new`, the operator `delete` does not have a return value.

Let us consider the example on the right.

E overloading.cpp

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 class MyClass
5 {
6     int _mymember;
7
8 public:
9     MyClass()
10    {
11         std::cout << "Constructor is called\n";
12     }
13
14     ~MyClass()
15    {
16         std::cout << "Destructor is called\n";
17     }
18
19     void *operator new(size_t size)
20    {
21         std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22         void *p = malloc(size);
23
24         return p;
25     }
26
27     void operator delete(void *p)
28    {
29         std::cout << "delete: Memory is freed again " << std::endl;
30         free(p);
31     }
32 }
```

root@efb13975c180:/home/v

root@efb13975c180:/home/workspace#

## overloading.cpp

```
21     std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22     void *p = malloc(size);
23
24     return p;
25 }
26
27 void operator delete(void *p)
28 {
29     std::cout << "delete: Memory is freed again " << std::endl;
30     free(p);
31 }
32 };
33
34 int main()
35 {
36     MyClass *p = new MyClass();
37     delete p;
38 }
```

In the code to the right, both the `new` and the `delete` operator are overloaded. In `new`, the size of the class object in bytes is printed to the console. Also, a block of memory of that size is allocated on the heap and the pointer to this block is returned. In `delete`, the block of memory is freed again. The console output of this example looks as follows:

```
new: Allocating 4 bytes of memory
Constructor is called
Destructor is called
delete: Memory is freed again
```

As can be seen from the order of text output, memory is instantiated in `new` before the constructor is called, while the order is reversed for the destructor and the call to `delete`.

## Quiz :

What will happen to the console output of the overloaded new operator when the data type of `_mymember` is changed from int to double?

1. Nothing.
2. There will be a memory leak as not enough memory is allocated on the heap.
3. The output will show a changed size in bytes (8 instead of 4).

**HIDE SOLUTION**

The correct answer is 3.

## Overloading new[] and delete[]

In addition to the `new` and `delete` operators we have seen so far, we can use the following code to create an array of objects:

```
void* operator new[](size_t size);
void operator delete[](void*);
```

Let us consider the example on the right, which has been slightly modified to allocate an array of objects instead of a single one.

overloading\_array.cpp

```
1 #include <iostream>
2 #include <stdlib.h>
3
4 class MyClass
5 {
6     int _mymember;
7
8 public:
9     MyClass()
10    {
11         std::cout << "Constructor is called\n";
12     }
13
14 ~MyClass()
15    {
16        std::cout << "Destructor is called\n";
17    }
18
19 void *operator new[](size_t size)
20 {
21     std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22     void *p = malloc(size);
23
24     return p;
25 }
26
```

root@efb13975c180:/home/v

root@efb13975c180:/home/workspace# []

## overloading\_array.cpp

```
21     std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22     void *p = malloc(size);
23
24     return p;
25 }
26
27 void operator delete[](void *p)
28 {
29     std::cout << "delete: Memory is freed again " << std::endl;
30     free(p);
31 }
32 };
33
34 int main()
35 {
36     MyClass *p = new MyClass[3]();
37     delete[] p;
38 }
```

In main, we are now creating an array of three objects of MyClass. Also, the overloaded `new` and `delete` operators have been changed to accept arrays. Let us take a look at the console output:

```
new: Allocating 20 bytes of memory
Constructor is called
Constructor is called
Constructor is called
Destructor is called
Destructor is called
Destructor is called
Destructor is called
delete: Memory is freed again
```

Interestingly, the memory requirement is larger than expected: With `new`, the block size was 4 bytes, which is exactly the space required for a single integer. Thus, with three integers, it should now be 12 bytes instead of 20 bytes. The reason for this is the memory allocation overhead that the compiler needs to keep track of the allocated blocks of memory - which in itself consumes memory. If we change the above call to e.g. `new MyClass[100]()`, we will see that the overhead of 8 bytes does not change:

```
new: Allocating 408 bytes of memory
Constructor is called

Destructor is called
delete: Memory is freed again
```

```
21     std::cout << "new: Allocating " << size << " bytes of memory" << std::endl;
22     void *p = malloc(size);
23
24     return p;
25 }
26
27 void operator delete[](void *p)
28 {
29     std::cout << "delete: Memory is freed again " << std::endl;
30     free(p);
31 }
32 };
33
34 int main()
35 {
36     MyClass *p = new MyClass[3]();
37     delete[] p;
38 }
```

```
root@efb13975c180:/home/v
```

```
root@efb13975c180:/home/workspace# []
```

## Reasons for overloading `new` and `delete`

Now that we have seen how to overload the `new` and `delete` operators, let us summarize the major scenarios where it makes sense to do this:

1. The overloaded `new` operator function allows to **add additional parameters**. Therefore, a class can have multiple overloaded `new` operator functions. This gives the programmer more **flexibility** in customizing the memory allocation for objects.
2. Overloaded the `new` and `delete` operators provides an easy way to **integrate a mechanism similar to garbage collection** capabilities (such as in Java), as we will shortly see later in this course.
3. By adding **exception handling capabilities** into `new` and `delete`, the code can be made more robust.
4. It is very easy to add customized behavior, such as overwriting deallocated memory with zeros in order to increase the security of critical application data.



Watch later

Share



#### Information

About  
News  
Tool Suite  
Supported Platforms  
The Developers

#### Source Code

Current Releases  
Release Archive  
Variants / Patches  
Code Repository  
Valkyrie / GUIs

#### Documentation

Table of Contents  
Quick Start  
FAQ  
User Manual  
Download Manual  
Research Papers  
Books

#### Contact

Mailing Lists and IRC  
Bug Reports  
Feature Requests  
Contact Summary  
Commercial Support

#### How to Help

Contributing  
Project Suggestions

#### Gallery

Projects / Users  
Press / Media  
Awards  
Surveys  
Artwork / Clothing



Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.

The Valgrind distribution currently includes six production-quality tools: a memory error detector, two thread error detectors, a cache and branch-prediction profiler, a call-graph generating cache and branch-prediction profiler, and a heap profiler. It also includes three experimental tools: a stack/global array overrun detector, a second heap detector, and a tool to help analyze how heap blocks are used, and a SimPoint basic block vector generator. It runs on the following platforms: X86/Linux, AMD64/Linux, ARM/Linux, ARM64/Linux, PPC32/Linux, PPC64/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, X86/Solaris, AMD64/Solaris, ARM/Android (2.3.x and later), ARM64/Android, X86/Android, X86/Darwin and AMD64/Darwin (Mac OS X 10.12).

Valgrind is [Open Source / Free Software](#), and is freely available under the [GNU General Public License, version 2](#).

#### Recent News

- Valgrind source code repository migrated from Subversion to git SCM at [sourceware.org](#).
- 14 April 2019: valgrind-3.15.0 is available. This release supports: X86/Linux, AMD64/Linux, ARM32/Linux, ARM64/Linux, PPC32/Linux, PPC64BE/Linux, PPC64LE/Linux, S390X/Linux, MIPS32/Linux, MIPS64/Linux, S390X/Linux, ARM/Android, ARM64/Android, MIPS32/Android, X86/Android, X86/Solaris, AMD64/Solaris, X86/MacOSX 10.12 and X86/Windows 10.

Copyright © 2000-2019 Valgrind™ Development

Hosting kindly donated by Mythic Beasts  
Best viewed with Aimal Browser

On this website, you will find a short overview of



2:33 / 5:26



YouTube



Valgrind Tool Suite

# ND213 C03 L03 04.2 Typical Memory Management Problems SC

Watch later Share

The Valgrind distribution includes the following debugging and profiling tools:

- Memcheck
- Cachegrind
- Callgrind
- Massif
- Helgrind
- DRD
- DHAT
- Experimental Tools
- Other Tools

## Memcheck

Memcheck detects memory-management problems, and is aimed primarily at C and C++ programs. When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted. As a result, Memcheck can detect if your program:

- Accesses memory it shouldn't (areas not yet allocated, areas that have been freed, areas past the end of heap blocks, inaccessible areas of the stack).
- Uses uninitialized values in dangerous ways.
- Leaks memory.
- Does bad frees of heap blocks (double frees, mismatched frees).
- Passes overlapping source and destination memory blocks to memcpy() and related functions.

Memcheck reports these errors as soon as they occur, giving the source line number at which it occurred, and also a stack trace of the functions called to reach that line. Memcheck tracks addressability at the byte-level, and initialisation of values at the bit-level. As a result, it can detect the use of single uninitialized bits, and does not report spurious errors on benchmarks. Memcheck runs programs about 10–30x slower than normal.

## Cachegrind

Cachegrind is a cache profiler. It performs detailed simulation of the L1, D1 and L2 caches in your CPU and so can accurately pinpoint the sources of cache misses in your code. It identifies the number of cache misses, memory references and instructions executed for each line of source code, with per-function, per-module and whole-program summaries. It is available in any language. Cachegrind runs programs about 20–100x slower than normal.

## Callgrind

Callgrind, by Josef Weidendorfer, is an extension to Cachegrind. It provides all the information that Cachegrind does, plus extra information about callgraphs. It was folded into the main Valgrind distribution in version 3.2.0. Available separately is an amazing visualisation tool, KCachegrind, which gives a much better overview of the data that Callgrind collects; it is available in any language. Callgrind runs programs about 20–100x slower than normal.

## Massif

Massif is a heap profiler. It performs detailed heap profiling by taking regular snapshots of a program's heap. It produces a graph showing heap usage over time, including information about which parts of the program are responsible for the most memory allocations. The graph is supplemented by a text or HTML file that includes more information for determining which parts of the program are being allocated. Massif runs programs about 20x slower than normal.

## Helgrind

Helgrind is a thread debugger which finds data races in multithreaded programs. It looks for memory locations which are accessed by more than one (POSIX p-)thread, but for which no consistently used (pthread\_mutex\_) lock can be found. Such locations are indicative of missing synchronisation between threads, and could cause hard-to-find timing-dependent bugs. Helgrind is available in any language. It is a somewhat experimental tool, so your feedback is especially welcome here.

## DRD

DRD is a tool for detecting errors in multithreaded C and C++ programs. The tool works for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX threading primitives. While Helgrind can detect locking order violations, for most programs DRD needs less memory to perform its analysis.

## Lackey, Nulgrind

Lackey and Nulgrind are also included in the Valgrind distribution. They don't do very much, and are there for testing and demonstrative purposes.

## DHAT

DHAT is a tool for examining how programs use their heap allocations. It tracks the allocated blocks, and inspects every memory access to find which block, if any, it is to. It comes with a GUI to facilitate exploring the profile results.

## Experimental Tools

### BBV

A basic block is a linear section of code with one entry point and one exit point. A basic block vector (BBV) is a list of all basic blocks visited during program execution, and a count of how many times each basic block was run.

BBV is a tool that generates basic block vectors for use with the other tools.

### SGCheck

SGCheck is a tool for finding overruns of stack and global arrays. It works by using a heuristic approach derived from an observation about the likely forms of stack and global array accesses.

Other Tools

2:57 / 5:26

CC HD YouTube

ND213 C03 L03 04.2 Typical Memory Management Problems SC

```
1
2 int main()
3 {
4     int *pInt = new int[10];
5
6     return 0;
7 }
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

4: BUILD and LAUNCH MM-+

```
bash-3.2$ ls
bin    src
bash-3.2$ valgrind
```

it's done by simply typing valgrind here,

CC HD YouTube

ND213 C03 L03 04.2 Typical Memory Management Problems SC

```
1
2     int main()
3     {
4         int *pInt = new int[10];
5
6         return 0;
7     }
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 4: BUILD and LAUNCH MM-+

```
bash-3.2$ ls
bin    src
bash-3.2$ valgrind --leak-check=full --show-leaks=kinds=all --track-origins=yes --verbose --log-file=./bin/valgrind-out.txt ./bin/memory_leaks_debugging
valgrind: Unknown option: --show-leaks=kinds=all
valgrind: Use --help for more information or consult the user manual.
bash-3.2$ valgrind --leak-check=full --show-leak=kinds=all --track-origins=yes --verbose --log-file=./bin/valgrind-out.txt ./bin/memory_leaks_debugging
bash-3.2$
```

CC HD YouTube

ND213 C03 L03 04.3 Typical Memory Management Problems SC

```
1
2 int main()
3 {
4     int *pInt = new int[10];
5
6     return 0;
7 }
```

valgrind-out.txt

```
==85311== by 0x1006D055F: __objc_personality_v0 (in /usr/lib/libobjc.A.dylib)
==85311== by 0x10000847A: dyld::notifyBatchPartial(dyld_image_states, bool, char const* (*)(dyld_image_states, unsigned int, dyld_image_info const*), bool, bool) (in /usr/lib/dyld)
==85311== by 0x10000862D: dyld::registerObjCNotifiers(void (*)(unsigned int, char const* const*, mach_header const* const*), void (*)(char const*, mach_header const* const*), void (*)(char const*, mach_header const* const*))
(in /usr/lib/dyld)
==85311== by 0x100318A26: _dyld_objc_notify_register (in /usr/lib/system/libdyld.dylib)
==85311== by 0x1006BD233: environ_init (in /usr/lib/libobjc.A.dylib)
==85311== by 0x1002AFE35: _os_object_init (in /usr/lib/system/libdispatch.dylib)
==85311== by 0x1002B8AD1: libdispatch_init (in /usr/lib/system/libdispatch.dylib)
==85311== by 0x1001939C4: libSystem_initializer (in /usr/lib/libSystem.B.dylib)
==85311== by 0x10001B591: ImageLoaderMachO::doModInitFunctions(ImageLoader::LinkContext const&) (in /usr/lib/dyld)
==85311== by 0x10001B797: ImageLoaderMachO::doInitialization(ImageLoader::LinkContext const&) (in /usr/lib/dyld)
==85311==
==85311== LEAK SUMMARY:
==85311==   definitely lost: 40 bytes in 1 blocks
==85311==   indirectly lost: 0 bytes in 0 blocks
==85311==   possibly lost: 72 bytes in 3 blocks
==85311==   still reachable: 200 bytes in 6 blocks
==85311==   suppressed: 18,985 bytes in 160 blocks
==85311==
==85311== ERROR SUMMARY: 428 errors from 30 contexts (suppressed: 6 from 6)
==85311==
==85311== 1 errors in context 1 of 30:
==85311== Conditional jump or move depends on uninitialized value(s)
==85311== at 0x10066BC1F: xpc_uint64_create (in /usr/lib/system/libxpc.dylib)
==85311== by 0x10066BBAD: _xpc_collect_images (in /usr/lib/system/libxpc.dylib)
==85311== by 0x10066B01E: libxpc_initializer (in /usr/lib/system/libxpc.dylib)
==85311== by 0x1001939C9: libSystem_initializer (in /usr/lib/libSystem.B.dylib)
==85311== by 0x10001B591: ImageLoaderMachO::doModInitFunctions(ImageLoader::LinkContext const&) (in /usr/lib/dyld)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 4: BUILD and LAUNCH MM- □ +

```
bash-3.2$ ls
bin    src
bash-3.2$ valgrind --leak-check=full --show-leaks=kinds=all --track-origins=yes --verbose --log-file=./bin/valgrind-out.txt ./bin/memory_leaks_debugging
valgrind: Unknown option: --show-leaks=kinds=all
valgrind: Use --help for more information or consult the user manual.
bash-3.2$ valgrind --leak-che
which has been found by Valgrind to be definitely lost.
```

Watch later Share

## Overview of memory management problems

One of the primary advantages of C++ is the flexibility and control of resources such as memory it gives to the programmer. This advantage is further amplified by a significant increase in the performance of C++ programs compared to other languages such as Python or Java.

However, these advantages come at a price as they demand a high level of experience from the programmer. As Bjarne Stroustrup put it so elegantly:

*"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off".*

In this chapter, we will look at a collection of typical errors in memory management that you need to watch out for.

**1. Memory Leaks** Memory leaks occur when data is allocated on the heap at runtime, but not properly deallocated. A program that forgets to clear a memory block is said to have a memory leak - this may be a serious problem or not, depending on the circumstances and on the nature of the program. For a program that runs, computes something, and quits immediately, memory leaks are usually not a big concern. Memory leaks are mostly problematic for programs that run for a long time and/or use large data structures. In such a case, memory leaks can gradually fill the heap until allocation requests can no longer be properly met and the program stops responding or crashes completely. We will look at an example further down in this section.

**2. Buffer Overruns** Buffer overruns occur when memory outside the allocated limits is overwritten and thus corrupted. One of the resulting problems is that this effect may not become immediately visible. When a problem finally does occur, cause and effect are often hard to discern. It is also sometimes possible to inject malicious code into programs in this way, but this shall not be discussed here.

In this example, the allocated stack memory is too small to hold the entire string, which results in a segmentation fault:

```
char str[5];
strcpy(str, "BufferOverrun");
printf("%s", str);
```

**3. Uninitialized Memory** Depending on the C++ compiler, data structures are sometimes initialized (most often to zero) and sometimes not. So when allocating memory on the heap without proper initialization, it might sometimes contain garbage that can cause problems.

Generally, a variable will be automatically initialized in these cases:

- it is a class instance where the default constructor initializes all primitive types
- array initializer syntax is used, such as `int a[10] = {}`
- it is a global or extern variable
- it is defined `static`

The behavior of the following code is potentially undefined:

```
int a;
int b=a*42;
printf("%d", b);
```

**4. Incorrect pairing of allocation and deallocation** Freeing a block of memory more than once will cause a program to crash. This can happen when a block of memory is freed that has never been allocated or has been freed before. Such behavior could also occur when improper pairings of allocation and deallocation are used such as using `malloc()` with `delete` or `new` with `free()`.

In this first example, the wrong `new` and `delete` are paired

```
double *pDbl=new double[5];
delete pDbl;
```

In this second example, the pairing is correct but a double deletion is performed:

```
char *pChr=new char[5];
delete[] pChr;
delete[] pChr;
```

**5. Invalid memory access** This error occurs when trying to access a block of heap memory that has not yet or has already been deallocated.

In this example, the heap memory has already been deallocated at the time when `strcpy()` tries to access it:

```
char *pStr=new char[25];
delete[] pStr;
strcpy(pStr, "Invalid Access");
```

Even experienced developers sometimes make mistakes that cannot be discovered at first glance.

Instead of spending a lot of time searching, it makes sense for C and C++ programmers to use helper tools to perform automatic analyses of their code.

In this section, we will look at *Valgrind*, a free software for Linux and Mac that is able to automatically detect memory. Windows programmers can for example use the Visual Studio debugger and C Run-time Library (CRT) to detect and identify memory leaks. More information on how to do this can be found here: [Find memory leaks with the CRT Library - Visual Studio | Microsoft Docs](#)

With recent versions of MacOS, occasional difficulties have been reported with installing Valgrind. A working version for MacOS Mojave can be downloaded from GitHub via Homebrew: [GitHub - sowson/valgrind: Experimental Version of Valgrind for macOS 10.14.6 Mojave](#)

Valgrind is a framework that facilitates the development of tools for the dynamic analysis of programs. Dynamic analysis examines the behavior of a program at runtime, in contrast to static analysis, which often checks programs for various criteria or potential errors at the source code level before, during, or after translation. More information on Valgrind can be found here: [Valgrind: About](#)

The Memcheck tool within Valgrind can be used to detect typical errors in programs written in C or C++ that occur in connection with memory management. It is probably the best-known tool in the Valgrind suite, and the name Valgrind is often used as a synonym for Memcheck.

The following code generates a memory leak as the integer array has been allocated on the heap but the deallocation has been forgotten by the programmer:

The array of integers on the heap to which `pInt` is pointing has a size of `10 * sizeof(int)`, which is 40 bytes. Let us now use Valgrind to search for this leak.

After compiling the `memory_leaks_debugging.cpp` code file on the right to `a.out`, the terminal can be used to start Valgrind with the following command:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --log-file=/home/workspace/valgrind-...
```

Let us look at the call parameters one by one:

- **--leak-check** : Controls the search for memory leaks when the client program finishes. If set to `summary`, it says how many leaks occurred. If set to `full`, each individual leak will be shown in detail.
- **--show-leak-kinds** : controls the set of leak kinds to show when `--leak-check=full` is specified. Options are `definite`, `indirect`, `possible`, `reachable`, `all` and `none`
- **--track-origins** : can be used to see where uninitialised values come from.

You can read the file into the terminal with:

```
cat valgrind-out.txt
```

In the following, a (small) excerpt of the `valgrind-out.txt` log file is given:

```
==952== 40 bytes in 1 blocks are definitely lost in loss record 18 of 45
...
==952==    by 0x10019A377: operator new(unsigned long) (in /usr/lib/libc++abi.dylib)
...
==952==    by 0x100000F8A: main (memory_leaks_debugging.cpp:12)
...
==952== LEAK SUMMARY:
==952==    definitely lost: 40 bytes in 1 blocks
==952==    indirectly lost: 0 bytes in 0 blocks
==952==    possibly lost: 72 bytes in 3 blocks
==952==    still reachable: 200 bytes in 6 blocks
==952==    suppressed: 18,876 bytes in 160 blocks
```

As expected, the memory leak caused by the omitted deletion of the array of 10 integers in the code sample above shows up in the leak summary. Additionally, the exact position where the leak occurs in the code (line 12) can also be seen together with the responsible call which caused the leak.

This short introduction into memory leak search is only an example of how powerful analysis tools such as Valgrind can be used to detect memory-related problems in your code.

## Default copying

Resource management is one of the primary responsibilities of a C++ programmer.

Among resources such as multi-threaded locks, files, network and database connections this also includes memory. The common denominator in all of these examples is that access to the resource is often managed through a handle such as a pointer. Also, after the resource has been used and is no longer, it must be released again so that it available for re-use by someone else.

In C++, a common way of safely accessing resources is by wrapping a manager class around the handle, which is initialized when the resource is acquired (in the class constructor) and released when it is deleted (in the class destructor). This concept is often referred to as *Resource Acquisition is Initialization (RAII)*, which we will discuss in greater depth in the next concept. One problem with this approach though is that copying the manager object will also copy the handle of the resource. This allows two objects access to the same resource - and this can mean trouble.

Consider the example on the right of managing access to a block of heap memory.

The class `MyClass` has a private member, which is a pointer to a heap-allocated integer. Allocation is performed in the constructor, deallocation is done in the destructor. This means that the memory block of size `sizeof(int)` is allocated when the objects `myClass1` and `myClass2` are created on the stack and deallocated when their scope is left, which happens at the end of the main. The difference between `myClass1` and `myClass2` is that the latter is instantiated using the copy constructor, which duplicates the members in `myClass1` - including the pointer to the heap memory where `_myInt` resides.

The output of the program looks like the following:

```
Own address on the stack is 0x7ffefbf670
Managing memory block on the heap at 0x100300060
Own address on the stack is 0x7ffefbf658
Managing memory block on the heap at 0x100300060
copy_constructor_1(87582,0x1000a95c0) malloc: *** error for object
0x100300060: pointer being freed was not allocated
```

Note that in the workspace, the error will read:

```
*** Error in './a.out': double free or corruption (fasttop):
0x0000000001133c20 ***
```

```
1 #include <iostream>
2
3 class MyClass
4 {
5 private:
6     int *_myInt;
7
8 public:
9     MyClass()
10    {
11         _myInt = (int *)malloc(sizeof(int));
12    }
13 ~MyClass()
14    {
15     free(_myInt);
16    }
17 void printOwnAddress() { std::cout << "Own address on the stack is " << this << std::endl; }
18 void printMemberAddress() { std::cout << "Managing memory block on the heap at " << _myInt << std::endl; }
19 }
20
21 int main()
22 {
23     // instantiate object 1
24     MyClass myClass1;
25     myClass1.printOwnAddress();
26     myClass1.printMemberAddress();
```

```
root@77c8596d697e:/home/
```

```
root@77c8596d697e:/home/workspace#
```

when the objects `myClass1` and `myClass2` are created on the stack and deallocated when their scope is left, which happens at the end of the main. The difference between `myClass1` and `myClass2` is that the latter is instantiated using the copy constructor, which duplicates the members in `myClass1` - including the pointer to the heap memory where `_myInt` resides.

The output of the program looks like the following:

```
Own address on the stack is 0x7ffefbf670
Managing memory block on the heap at 0x100300060
Own address on the stack is 0x7ffefbf658
Managing memory block on the heap at 0x100300060
copy_constructor_1(87582,0x1000a95c0) malloc: *** error for object
0x100300060: pointer being freed was not allocated
```

Note that in the workspace, the error will read:

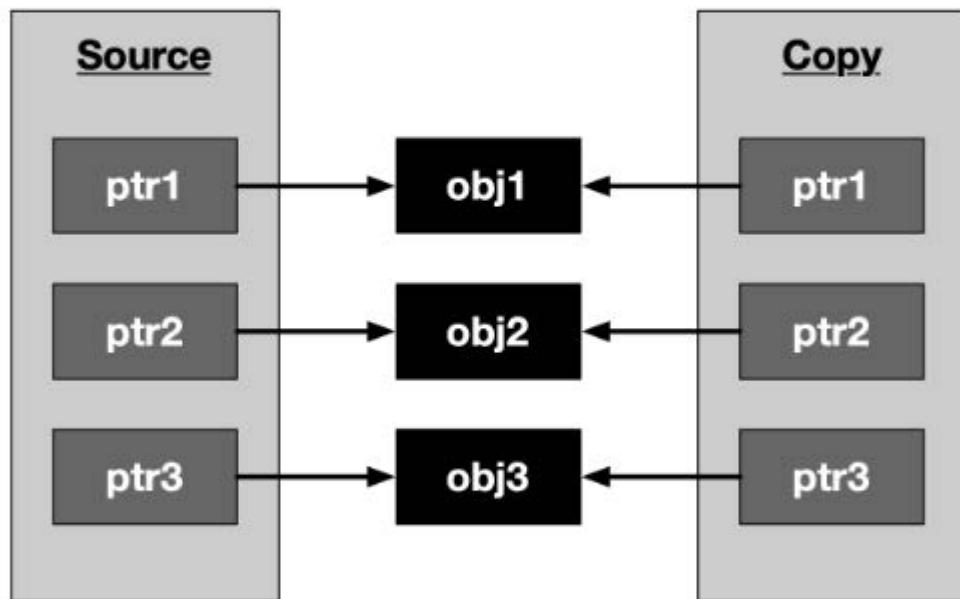
```
*** Error in './a.out': double free or corruption (fasttop):
0x0000000001133c20 ***
```

From the output we can see that the stack address is different for `myClass1` and `myClass2` - as was expected. The address of the managed memory block on the heap however is identical. This means that when the first object goes out of scope, it releases the memory resource by calling `free` in its destructor. The second object does the same - which causes the program to crash as the pointer is now referencing an invalid area of memory, which has already been freed.

```
21 int main()
22 {
23     // instantiate object 1
24     MyClass myClass1;
25     myClass1.printOwnAddress();
26     myClass1.printMemberAddress();
27
28     // copy object 1 into object 2
29     MyClass myClass2(myClass1); // copy constructor
30     myClass2.printOwnAddress();
31     myClass2.printMemberAddress();
32
33     return 0;
34 }
```

The default behavior of both copy constructor and assignment operator is to perform a *shallow copy* as with the example above. The following figure illustrates the concept:

### Shallow Copy



Fortunately, in C++, the copying process can be controlled by defining a tailored copy constructor as well as a copy assignment operator. The copying process

copy constructor as well as a copy assignment operator. The copying process must be closely linked to the respective resource release mechanism and is often referred to as *copy-ownership policy*. Tailoring the copy constructor according to your memory management policy is an important choice you often need to make when designing a class. In the following, we will closely examine several well-known copy-ownership policies.

## No copying policy

The simplest policy of all is to forbid copying and assigning class instances all together. This can be achieved by declaring, but not defining a private copy constructor and assignment operator (see `NoCopyClass1` below) or alternatively by making both public and assigning the `delete` operator (see `NoCopyClass2` below). The second choice is more explicit and makes it clearer to the programmer that copying has been actively forbidden. Let us have a look at a code example on the right that illustrates both cases.

On compiling, we get the following error messages:

```
error: calling a private constructor of class 'NoCopyClass1'  
    NoCopyClass1 copy1(original1);  
    NoCopyClass1 copy1b = original1;  
  
error: call to deleted constructor of 'NoCopyClass2'  
    NoCopyClass2 copy2(original2);  
    NoCopyClass2 copy2b = original2;
```

Both cases effectively prevent the original object from being copied or assigned. In the C++11 standard library, there are some classes for multi-threaded synchronization which use the no copying policy.

## Exclusive ownership policy

This policy states that whenever a resource management object is copied, the resource handle is transferred from the source pointer to the destination pointer. In the process, the source pointer is set to `nullptr` to make ownership exclusive. At any time, the resource handle belongs only to a single object, which is responsible for its deletion when it is no longer needed.

The code example on the right illustrates the basic idea of exclusive ownership.

The class `MyClass` overwrites both the copy constructor as well as the assignment operator. Inside, the handle to the resource `_myInt` is first copied from the source object and then set to null so that only a single valid handle exists. After copying, the new object is responsible for properly deleting the memory resource on the heap. The output of the program looks like the following:

```
resource allocated  
resource freed
```

As can be seen, only a single resource is allocated and freed. So by passing handles and invalidating them, we can implement a basic version of an exclusive ownership policy. However, this example is not the way exclusive ownership is handled in the standard template library. One problem in this implementation is that for a short time there are effectively two valid handles to the same resource - after the handle has been copied and before it is set to `nullptr`. In concurrent programs, this would cause a data race for the resource. A much better alternative to handle exclusive ownership in C++ would be to use move semantics, which we will discuss shortly in a very detailed lesson.

## Deep copying policy

With this policy, copying and assigning class instances to each other is possible without the danger of resource conflicts. The idea is to allocate proprietary memory in the destination object and then to copy the content to which the source object handle is pointing into the newly allocated block of memory. This way, the content is preserved during copy or assignment. However, this approach increases the memory demands and the uniqueness of the data is lost: After the deep copy has been made, two versions of the same resource exist in memory.

Let us look at an example in the code on the right.

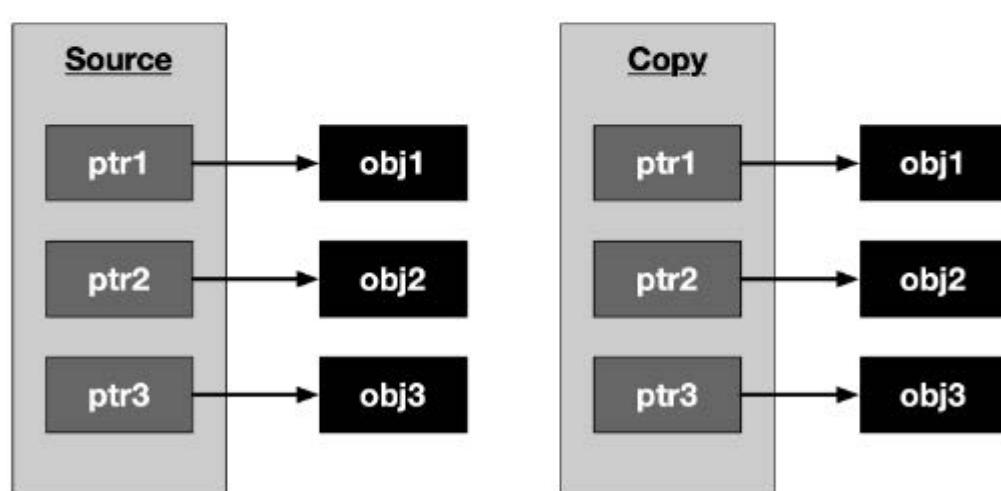
The deep-copy version of MyClass looks similar to the exclusive ownership policy: Both the assignment operator and the copy constructor have been overloaded with the source object passed by reference. But instead of copying the source handle (and then deleting it), a proprietary block of memory is allocated on the heap and the content of the source is copied into it.

The output of the program looks like the following:

```
resource allocated at address 0x100300060
resource allocated at address 0x100300070 with _myInt = 42
resource allocated at address 0x100300080 with _myInt = 42
resource freed at address 0x100300080
resource freed at address 0x100300070
resource freed at address 0x100300060
```

As can be seen, all copies have the same value of 42 while the address of the handle differs between `source`, `dest1` and `dest2`.

To conclude, the following figure illustrates the idea of a deep copy:



## Shared ownership policy

The last ownership policy we will be discussing in this course implements a shared ownership behavior. The idea is to perform a copy or assignment similar to the default behavior, i.e. copying the handle instead of the content (as with a shallow copy) while at the same time keeping track of the number of instances that also point to the same resource. Each time an instance goes out of scope, the counter is decremented. Once the last object is about to be deleted, it can safely deallocate the memory resource. We will see later in this course that this is the central idea of `unique_ptr`, which is a representative of the group of smart pointers.

The example on the right illustrates the principle.

Note that class `MyClass` now has a static member `_cnt`, which is incremented every time a new instance of `MyClass` is created and decremented once an instance is deleted. On deletion of the last instance, i.e. when `_cnt==0`, the block of memory to which the handle points is deallocated.

The output of the program is the following:

```
resource allocated at address 0x100300060
2 instances with handles to address 0x100300060 with _myInt = 42
3 instances with handles to address 0x100300060 with _myInt = 42
4 instances with handles to address 0x100300060 with _myInt = 42
instance at address 0x7fffeefbff6f8 goes out of scope with _cnt = 3
instance at address 0x7fffeefbff700 goes out of scope with _cnt = 2
instance at address 0x7fffeefbff718 goes out of scope with _cnt = 1
resource freed at address 0x100300060
```

As can be seen, the memory is released only once as soon as the reference counter reaches zero.

## The Rule of Three

In the previous examples we have taken a first look at several copying policies:

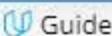
1. Default copying
2. No copying
3. Exclusive ownership
4. Deep copying
5. Shared ownership

In the first example we have seen that the default implementation of the copy constructor does not consider the "special" needs of a class which allocates and deallocates a shared resource on the heap. The problem with implicitly using the default copy constructor or assignment operator is that programmers are not forced to consider the implications for the memory management policy of their program. In the case of the first example, this leads to a segmentation fault and thus a program crash.

In order to properly manage memory allocation, deallocation and copying behavior, we have seen that there is an intricate relationship between destructor, copy constructor and copy assignment operator. To this end, the **Rule of Three** states that if a class needs to have an overloaded copy constructor, copy assignment operator, ~or~ destructor, then it must also implement the other two as well to ensure that memory is managed consistently. As we have seen, the copy constructor and copy assignment operator (which are often almost identical) control how the resource gets copied between objects while the destructor manages the resource deletion.

You may have noted that in the previous code example, the class `SharedCopy` does not implement the assignment operator. This is a violation of the **Rule of Three** and thus, if we were to use something like `destination3 = source` instead of `SharedCopy destination3(source)`, the counter variable would not be properly decremented.

The copying policies discussed in this chapter are the basis for a powerful concept in C++11 - smart pointers. But before we discuss these, we need to go into further detail on move semantics, which is a prerequisite you need to learn more about so you can properly understand the exclusive ownership policy as well as the Rule of Five, both of which we will discuss very soon. But before we discuss move semantics, we need to look into the concept of lvalues and rvalues in the next section.



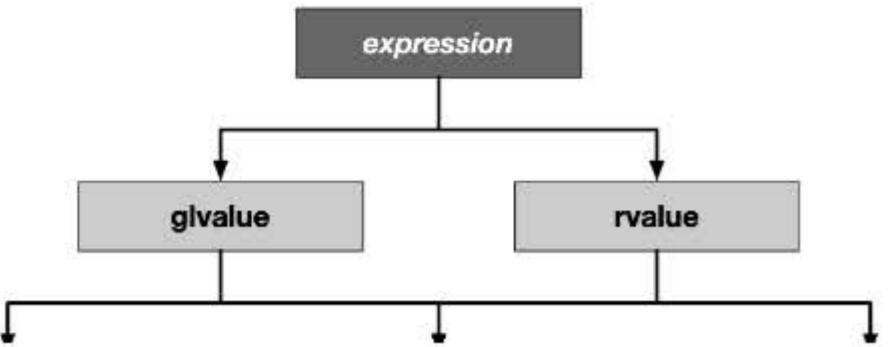
Guide

## What are lvalues and rvalues?

A good grasp of lvalues and rvalues in C++ is essential for understanding the more advanced concepts of rvalue references and motion semantics.

Let us start by stating that every expression in C++ has a type and belongs to a value category. When objects are created, copied or moved during the evaluation of an expression, the compiler uses these value expressions to decide which method to call or which operator to use.

Prior to C++11, there were only two value categories, now there are as many as five of them:



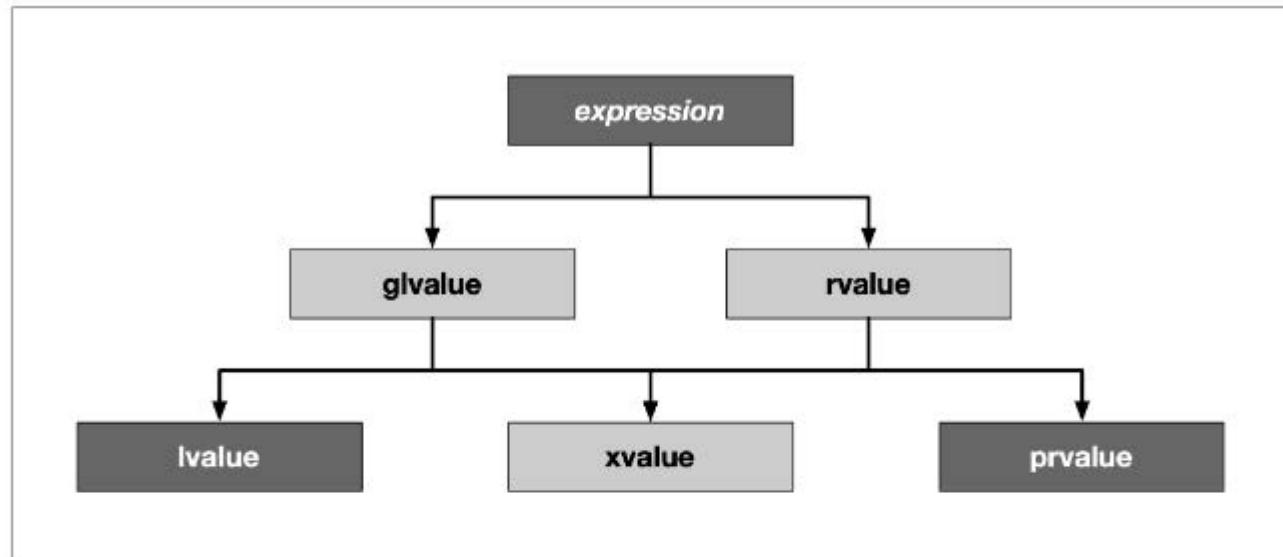
l\_and\_r\_value\_examples.cpp

```

1 int main()
2 {
3     // initialize some variables on the stack
4     int i, j, *p;
5
6     // correct usage of lvalues and rvalues
7
8     i = 42; // i is an lvalue and 42 is an rvalue
9
10    p = new int;
11    *p = i; // the dereferenced pointer is an lvalue
12    delete p;
13
14    ((i < 42) ? i : j) = 23; // the conditional operator returns an lvalue (either i or j)
15
16    // incorrect usage of lvalues and rvalues:
17    // 42 = i; // error ; the left operand must be an lvalue
18    // j * 42 = 23; // error ; the left operand must be an lvalue
19
20    return 0;
21 }
```

root@c57ae5d4e023:/home/1

root@c57ae5d4e023:/home/workspace#



To keep it short, we do not want to go into all categories, but limit ourselves to lvalues and prvalues:

- Lvalues have an address that can be accessed. They are expressions whose evaluation by the compiler determines the identity of objects or functions.
- Prvalues do not have an address that is accessible directly. They are temporary expressions used to initialize objects or compute the value of the operand of an operator.

For the sake of simplicity and for compliance with many tutorials, videos and books

about the topic, let us refer to *prvalues* as *rvalues* from here on.

The two characters `l` and `r` are originally derived from the perspective of the assignment operator `=`, which always expects a rvalue on the right, and which it assigns to a lvalue on the left. In this case, the `l` stands for left and `r` for right:

```
int i = 42; // lvalue = rvalue;
```

With many other operators, however, this right-left view is not entirely correct. In more general terms, an lvalue is an entity that points to a specific memory location. An rvalue is usually a short-lived object, which is only needed in a narrow local scope. To simplify things a little, one could think of lvalues as *named containers* for rvalues.

In the example above, the value `42` is an rvalue. It does not have a specific memory address which we know about. The rvalue is assigned to a variable `i` with a specific memory location known to us, which is what makes it an lvalue in this example.

Using the address operator `&` we can generate an lvalue from an rvalue and assign it to another lvalue:

```
int *j = &i;
```

In this small example, the expression `&i` generates the address of `i` as an rvalue and assigns it to `j`, which is an lvalue now holding the memory location of `i`.

The code on the right illustrates several examples of lvalues and rvalues:

## Lvalue references

An lvalue reference can be considered as an alternative name for an object. It is a reference that binds to an lvalue and is declared using an optional list of specifiers (which we will not further discuss here) followed by the reference declarator `&`. The short code sample on the right declares an integer `i` and a reference `j` which can be used as an alias for the existing object.

The output of the program is

```
i = 3, j = 3
```

We can see that the lvalue reference `j` can be used just as `i` can. A change to either `i` or `j` will affect the same memory location on the stack.

### l\_value\_references.cpp

```
1 #include <iostream>
2
3 int main()
4 {
5     int i = 1;
6     int &j = i;
7     ++i;
8     ++j;
9
10    std::cout << "i = " << i << ", j = " << j << std::endl;
11
12    return 0;
13 }
```

```
root@c57ae5d4e023: /home/
```

```
root@c57ae5d4e023:/home/workspace#
```

One of the primary use-cases for lvalue references is the pass-by-reference semantics in function calls as in the example on the right.

The function `myFunction` has an lvalue reference as a parameter, which establishes an alias to the integer `i` which is passed to it in `main`.

```
1 #include <iostream>
2
3 void myFunction(int &val)
4 {
5     ++val;
6 }
7
8 int main()
9 {
10    int i = 1;
11    myFunction(i);
12
13    std::cout << "i = " << i << std::endl;
14
15    return 0;
16 }
```

```
root@c57ae5d4e023: /home/
```

```
root@c57ae5d4e023:/home/workspace#
```

## Rvalue references

You already know that an rvalue is a temporary expression which is - among other use-cases, a means of initializing objects. In the call

`int i = 42`, 42 is the rvalue.

Let us consider an example similar to the last one, shown on the right.

As before, the function `myFunction` takes an lvalue reference as its argument. In `main`, the call `myFunction(j)` works just fine while `myFunction(42)` as well as `myFunction(j+k)` produces the following compiler error on Mac:

```
candidate function not viable: expects an l-value for 1st argument
```

and the following error in the workspace with g++:

```
error: cannot bind non-const lvalue reference of type  
'int&' to an rvalue of type 'int'
```

While the number `42` is obviously an rvalue, with `j+k` things might

### r\_value\_references.cpp

```
1 #include <iostream>  
2  
3 void myFunction(int &val)  
4 {  
5     std::cout << "val = " << val << std::endl;  
6 }  
7  
8 int main()  
9 {  
10    int j = 42;  
11    myFunction(j);  
12  
13    myFunction(42);  
14  
15    int k = 23;  
16    myFunction(j+k);  
17  
18    return 0;  
19 }
```

```
root@c57ae5d4e023:/home/|
```

```
root@c57ae5d4e023:/home/workspace# |
```

While the number 42 is obviously an rvalue, with  $j+k$  things might not be so obvious, as  $j$  and  $k$  are variables and thus lvalues. To compute the result of the addition, the compiler has to create a temporary object to place it in - and this object is an rvalue.

Since C++11, there is a new type available called *rvalue reference*, which can be identified from the double ampersand `&&` after a type name. With this operator, it is possible to store and even modify an rvalue, i.e. a temporary object which would otherwise be lost quickly.

But what do we need this for? Before we look into the answer to this question, let us consider the example on the right.

## r\_value\_references\_2.cpp

```
1 #include <iostream>
2
3 int main()
4 {
5     int i = 1;
6     int j = 2;
7     int k = i + j;
8     int &&l = i + j;
9
10    std::cout << "k = " << k << ", l = " << l << std::endl;
11
12    return 0;
13 }
```

```
root@c57ae5d4e023: /home/
```

```
root@c57ae5d4e023:/home/workspace#
```

After creating the integers `i` and `j` on the stack, the sum of both is added to a third integer `k`. Let us examine this simple example a little more closely. In the first and second assignment, `i` and `j` are created as lvalues, while `1` and `2` are rvalues, whose value is copied into the memory location of `i` and `j`. Then, a third lvalue, `k`, is created. The sum `i+j` is created as an rvalue, which holds the result of the addition before being copied into the memory location of `k`. This is quite a lot of copying and holding of temporary values in memory. With an rvalue reference, this can be done more efficiently.

The expression `int &&1` creates an rvalue reference, to which the address of the temporary object is assigned, that holds the result of the addition. So instead of first creating the rvalue `i+j`, then copying it and finally deleting it, we can now hold the temporary object in memory. This is much more efficient than the first approach, even though saving a few bytes of storage in the example might not seem like much at first glance. One of the most important aspects of rvalue references is that they pave the way for *move semantics*, which is a mighty technique in modern C++ to optimize memory usage and

## r\_value\_references\_2.cpp

```
1 #include <iostream>
2
3 int main()
4 {
5     int i = 1;
6     int j = 2;
7     int k = i + j;
8     int &&1 = i + j;
9
10    std::cout << "k = " << k << ", 1 = " << 1 << std::endl;
11
12    return 0;
13 }
```

```
[root@c57ae5d4e023: /home/]
```

```
root@c57ae5d4e023:/home/workspace#
```

processing speed. Move semantics and rvalue references make it possible to write code that transfers resources such as dynamically allocated memory from one object to another in a very efficient manner and also supports the concept of exclusive ownership, as we will shortly see when discussing smart pointers. In the next section we will take a close look at move semantics and its benefits for memory management.



## Rvalue references and std::move

In order to fully understand the concept of smart pointers in the next lesson, we first need to take a look at a powerful concept introduced with C++11 called *move semantics*.

The last section on lvalues, rvalues and especially rvalue references is an important prerequisite for understanding the concept of moving data structures.

Let us consider the function on the right which takes an rvalue reference as its parameter.

The important message of the function argument of `myFunction` to the programmer is : The object that binds to the rvalue reference `&&val` is yours, it is not needed anymore within the scope of the caller (which is `main`). As discussed in the previous section on rvalue references, this is interesting from two perspectives:

1. Passing values like this **improves performance** as no temporary copy needs to be made anymore and
2. **ownership changes**, since the object the reference binds to has been abandoned by the caller and now binds to a handle which is available only to the receiver. This could not have been achieved with lvalue references as any change to the object that binds to the lvalue reference would also be visible on the caller side.

There is one more important aspect we need to consider: *rvalue references are themselves lvalues*. While this might seem confusing at first glance, it really is the mechanism that enables move semantics: A reference is always defined in a certain context (such as in the above example the variable `val`). Even though the object it refers to (the number `42`) may be disposable in the context it has been created (the `main` function), it is not disposable in the context of the reference. So within the scope of `myFunction`, `val` is an lvalue as it gives access to the memory location where the number 42 is stored.

Note however that in the above code example we cannot pass an lvalue to `myFunction`, because an rvalue reference cannot bind to an lvalue. The code

```
int i = 23;
myFunction(i)
```

would result in a compiler error. There is a solution to this problem though: The function `std::move` converts an lvalue into an rvalue (actually, to be exact, into an *xvalue*, which we will not discuss here for the sake of clarity), which makes it possible to use the lvalue as an argument for the function:



```
int i = 23;  
myFunction(std::move(i));
```

In doing this, we state that in the scope of `main` we will not use `i` anymore, which now exists only in the scope of `myFunction`. Using `std::move` in this way is one of the components of move semantics, which we will look into shortly. But first let us consider an example of the Rule of Three.

Let us consider the example to the right of a class which manages a block of dynamic memory and incrementally add new functionality to it. You will add the main function shown above later on in this notebook.

In this class, a block of heap memory is allocated in the constructor and deallocated in the destructor. As we have discussed before, when either destructor, copy constructor or copy assignment operator are defined, it is good practice to also define the other two (known as the **Rule of Three**). While the compiler would generate default versions of the missing components, these would not properly reflect the memory management strategy of our class, so leaving out the manual implementation is usually not advised.

So let us start with the copy constructor of `MyMovableClass`, which could look like the following:

```
MyMovableClass(const MyMovableClass &source) // 2 : copy
constructor
{
    _size = source._size;
    _data = new int[_size];
    *_data = *source._data;
    std::cout << "COPYING content of instance " << &source
<< " to instance " << this << std::endl;
}
```

Similar to an example in the section on copy semantics, the copy constructor takes an lvalue reference to the source instance, allocates a block of memory of the same size as in the source and then copies the data into its members (as a deep copy).

You can add this code to the `rule_of_three.cpp` file on the right.

Next, let us take a look at the copy assignment operator:

```
MyMovableClass &operator=(const MyMovableClass &source) // 3
: copy assignment operator
{
    std::cout << "ASSIGNING content of instance " << &source
<< " to instance " << this << std::endl;
    if (this == &source)
        return *this;
    delete[] _data;
    _data = new int[source._size];
    *_data = *source._data;
    _size = source._size;
    return *this;
}
```

You can add the code above to the `rule_of_three.cpp` file on the right.

The if-statement at the top of the above implementation protects against self-assignment and is standard boilerplate code for the user-defined assignment operator. The remainder of the code is more or less identical to the copy constructor, apart from returning a reference to the own instance using `this`.

You might have noticed that both copy constructor and assignment operator take a `const` reference to the source object as an argument, by which they promise that they won' (and can't) modify the content of source.



Page

6



of 17



We can now use our class to copy objects as shown in the following implementation of `main`:

```
int main()
{
    MyMovableClass obj1(10); // regular constructor
    MyMovableClass obj2(obj1); // copy constructor
    obj2 = obj1; // copy assignment operator

    return 0;
}
```



Add this code to the `rule_of_three.cpp` file on the right.

In the `main` above, the object `obj1` is created using the regular constructor of `MyMovableClass`. Then, both the copy constructor as well as the assignment operator are used with the latter one not creating a new object but instead assigning the content of `obj1` to `obj2` as defined by our copying policy.

The output of this textbook implementation of the Rule of Three looks like this:



CREATING instance of MyMovableClass at 0x7fffeefbff618  
allocated with size = 40 bytes

COPYING content of instance 0x7fffeefbff618 to instance  
0x7fffeefbff608

ASSIGNING content of instance 0x7fffeefbff618 to instance  
0x7fffeefbff608

DELETING instance of MyMovableClass at 0x7fffeefbff608

DELETING instance of MyMovableClass at 0x7fffeefbff618

## Limitations of Our Current Class Design

Let us now consider one more way to instantiate `MyMovableClass` object by using `createObject()` function. Add the following function definition to the `rule_of_three.cpp`, outside the scope of the class `MyMovableClass`:

```
MyMovableClass createObject(int size){  
    MyMovableClass obj(size); // regular constructor  
    return obj; // return MyMovableClass object by value  
}
```

Note that when a function returns an object by value, the compiler creates a temporary object as an rvalue. Let's call this function inside `main` to create an `obj4` instance, as follows:

```
int main(){  
    // call to copy constructor, (alternate syntax)  
    MyMovableClass obj3 = obj1;  
    // Here, we are instantiating obj3 in the same statement;  
    // hence the copy assignment operator would not be called.
```

```
MyMovableClass obj4 = createObject(10);
// createObject(10) returns a temporary copy of the object
as an rvalue, which is passed to the copy constructor.
```

```
/*
 * You can try executing the statement below as well
 * MyMovableClass obj4(createObject(10));
 */
return 0;
}
```

In the `main` above, the returned value of `createObject(10)` is passed to the copy constructor. The function `createObject()` returns an instance of `MyMovableClass` by value. In such a case, the compiler creates a temporary copy of the object as an rvalue, which is passed to the copy constructor.

### *A special call to copy constructor*

*Try compiling and then running the*

*rule\_of\_three.cpp to notice that*

*MyMovableClass obj4 = createObject(10);*

*would not print the cout statement of copy constructor on the console. This is because the copy constructor is called on the temporary object.*

In our current class design, while creating `obj4`, the data is dynamically allocated on the stack, which is then copied from the temporary object to its target destination. This means that **two expensive memory operations** are performed with the first occurring during the creation of the temporary rvalue and the second during the execution of the copy constructor. The similar two expensive memory operations would be performed with the assignment operator if we execute the following statement inside `main`:

```
MyMovableClass obj4 = createObject(10); // Don't write this  
statement if you have already written it before  
obj4 = createObject(10); // call to copy assignment operator
```



In the above call to copy assignment operator, it would first erase the memory of `obj4`, then reallocate it during the creation of the temporary object; and then copy the data from the temporary object to `obj4`.

From a performance viewpoint, this code involves far too many copies, making it inefficient - especially with large data structures. Prior to C++11, the proper solution in such a case was to simply avoid returning large data structures by value to prevent the expensive and unnecessary copying process. With C++11 however, there is a way we can optimize this and return even large data structures by value. The solution is the move constructor and the Rule of Five.

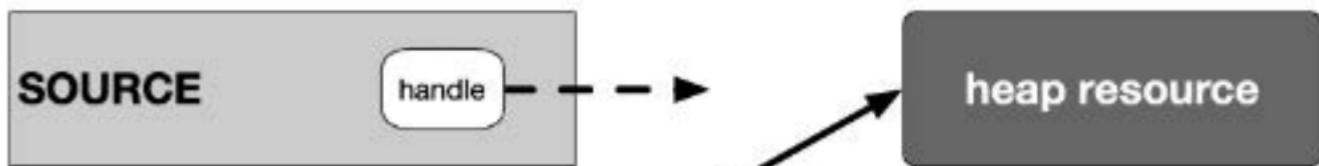
## The move constructor

The basic idea to optimize the code from the last example is to "steal" the rvalue generated by the compiler during the return-by-value operation and move the expensive data in the source object to the target object - not by copying it but by redirecting the data handles. Moving data in such a way is always cheaper than making copies, which is why programmers are highly encouraged to make use of this powerful tool.

The following diagram illustrates the basic principle of moving a resource from a source object to a destination object:



move



In order to achieve this, we will be using a construct called *move constructor*, which is similar to the copy constructor with the key difference being the re-use of existing data without unnecessarily copying it. In addition to the move constructor, there is also a move assignment operator, which we need to look at.

Just like the copy constructor, the move constructor builds an instance of a class using a source instance. The key difference between the two is that with the move constructor, the source instance will no longer be usable afterwards. Let us take a look at an implementation of the move constructor for our `MyMovableClass`:

```
MyMovableClass(MyMovableClass &&source) // 4 : move constructor
{
    std::cout << "MOVING (c'tor) instance " << &source << " to instance " << this << std::endl;
    _data = source._data;
    _size = source._size;
    source._data = nullptr;
    source._size = 0;
}
```

If you haven't already added it, you can add this code to the `rule_of_five.cpp` file to the right.

In this code, the move constructor takes as its input an rvalue reference to a `source` object of the same class. In doing so, we are able to use the object within the scope of the move constructor. As can be seen, the implementation copies the data handle from `source` to target and immediately invalidates `source` after copying is complete. Now, `this` is responsible for the data and must also release memory on destruction - the ownership has been successfully changed (or moved) without the need to copy the data on the heap.

The move assignment operator works in a similar way:

```
MyMovableClass &operator=(MyMovableClass &&source) // 5 : move assignment operator
{
    std::cout << "MOVING (assign) instance " << &source << " to instance " << this <<
std::endl;
    if (this == &source)
        return *this;

    delete[] _data;

    _data = source._data;
    _size = source._size;

    source._data = nullptr;
    source._size = 0;

    return *this;
}
```

As with the move constructor, the data handle is copied from `source` to target which is coming in as an rvalue reference again. Afterwards, the data members of `source` are invalidated. The rest of the code is identical with the copy constructor we have already implemented.

## The Rule of Five

By adding both the move constructor and the move assignment operator to our `MyMovableClass`, we have adhered to the **Rule of Five**. This rule is an extension of the Rule of Three which we have already seen and exists since the introduction of the C++11 standard. The Rule of Five is especially important in resource management, where unnecessary copying needs to be avoided due to limited resources and performance reasons. Also, all the STL container classes such as `std::vector` implement the Rule of Five and use move semantics for increased efficiency.

The Rule of Five states that if you have to write one of the functions listed below then you should consider implementing all of them with a proper resource management policy in place. If you forget to implement one or more, the compiler will usually generate the missing ones (without a warning) but the default versions might not be suitable for the purpose you have in mind. The five functions are:

1. **The destructor:** Responsible for freeing the resource once the object it belongs to goes out of scope.
2. **The assignment operator:** The default assignment operation performs a member-wise shallow copy, which does not copy the content behind the resource handle. If a deep copy is needed, it has to be implemented by the programmer.
3. **The copy constructor:** As with the assignment operator, the default copy constructor performs a shallow copy of the data members. If something else is needed, the

programmer has to implement it accordingly.

4. The **move constructor**: Because copying objects can be an expensive operation which involves creating, copying and destroying temporary objects, rvalue references are used to bind to an rvalue. Using this mechanism, the move constructor transfers the ownership of a resource from a (temporary) rvalue object to a permanent lvalue object.
5. The **move assignment operator**: With this operator, ownership of a resource can be transferred from one object to another. The internal behavior is very similar to the move constructor.

## When are move semantics used?

Now that we have seen how move semantics work, let us take a look at situations where they actually apply.

One of the primary areas of application are cases, where heavy-weight objects need to be passed around in a program. Copying these without move semantics can cause serious performance issues. The idea in this scenario is to create the object a single time and then "simply" move it around using rvalue references and move semantics.

A second area of application are cases where ownership needs to be transferred (such as with unique pointers, as we will soon see). The primary difference to shared references is that with move semantics we are not sharing anything but instead we are ensuring through a smart policy that only a single object at a time has access to and thus owns the resource.

Let us look at some code examples:

```
int main()
{
    MyMovableClass obj1(100), obj2(200); // constructor

    MyMovableClass obj3(obj1); // copy constructor

    MyMovableClass obj4 = obj1; // copy constructor

    obj4 = obj2; // copy assignment operator

    return 0;
}
```

If you compile and run this code, be sure to use the `-std=c++11` flag. The reasons for this will be explained below.

In the code above, in total, four instances of `MyMovableClass` are constructed here. While `obj1` and `obj2` are created using the conventional constructor, `obj3` is created using the copy constructor instead according to our implementation. Interestingly, even though the creation of `obj4` looks like an assignment, the compiler calls the copy constructor in this case. Finally, the last line calls the copy assignment operator. The output of the above main function looks like the following:



```
CREATING instance of MyMovableClass at 0x7fffeefbff718 allocated with  
size = 400 bytes
```

```
CREATING instance of MyMovableClass at 0x7fffeefbff708 allocated with  
size = 800 bytes
```

```
COPYING content of instance 0x7fffeefbff718 to instance 0x7fffeefbff6e8
```

```
COPYING content of instance 0x7fffeefbff718 to instance 0x7fffeefbff6d8
```

```
ASSIGNING content of instance 0x7fffeefbff708 to instance 0x7fffeefbff6d8
```

```
DELETING instance of MyMovableClass at 0x7fffeefbff6d8
```

```
DELETING instance of MyMovableClass at 0x7fffeefbff6e8
```

```
DELETING instance of MyMovableClass at 0x7fffeefbff708
```

```
DELETING instance of MyMovableClass at 0x7fffeefbff718
```

Note that the compiler has been called with the option

`-fno-elide-constructors` to turn off an optimization techniques called *copy elision*, which would make it harder to understand the various calls and the operations they entail. This technique is guaranteed to be used as of C++17, which is why we are also reverting to the C++11 standard for the remainder of this chapter using `-std=c++11`. Until now, no move operation has been performed yet as all of the above calls were involving lvalues.

Now consider the following `main` function instead:

```
int main()
{
    MyMovableClass obj1(100); // constructor

    obj1 = MyMovableClass(200); // move assignment operator

    MyMovableClass obj2 = MyMovableClass(300); // move constructor

    return 0;
}
```

In this version, we also have an instance of `MyMovableClass`, `obj1`. Then, a second instance of `MyMovableClass` is created as an rvalue, which is assigned to `obj1`. Finally, we have a second lvalue `obj2`, which is created by assigning it an rvalue object. Let us take a look at the output of the program:

```
CREATING instance of MyMovableClass at 0x7fffeefbff718 allocated with
size = 400 bytes

CREATING instance of MyMovableClass at 0x7fffeefbff708 allocated with
size = 800 bytes

MOVING (assign) instance 0x7fffeefbff708 to instance 0x7fffeefbff718

DELETING instance of MyMovableClass at 0x7fffeefbff708
```

```
CREATING instance of MyMovableClass at 0x7fffeefbff6d8 allocated with  
size = 1200 bytes
```

```
MOVING (c'tor) instance 0x7fffeefbff6d8 to instance 0x7fffeefbff6e8
```

```
DELETING instance of MyMovableClass at 0x7fffeefbff6d8
```

```
DELETING instance of MyMovableClass at 0x7fffeefbff6e8
```

```
DELETING instance of MyMovableClass at 0x7fffeefbff718
```

By looking at the stack addresses of the objects, we can see that the temporary object at `0x7fffeefbff708` is moved to `0x7fffeefbff718` using the move assignment operator we wrote earlier, because the instance `obj1` is assigned an rvalue. As expected from an rvalue, its destructor is called immediately afterwards. But as we have made sure to null its data pointer in the move constructor, the actual data will not be deleted. The advantage from a performance perspective in this case is that no deep-copy of the rvalue object needs to be made, we are simply redirecting the internal resource handle thus making an efficient shallow copy.

Next, another temporary instance with a size of 1200 bytes is created as a temporary object and "assigned" to `obj3`. Note that while the call looks like an assignment, the move constructor is called under the hood, making the call identical to `MyMovableClass obj2(MyMovableClass(300));`. By creating `obj3` in such a way, we are reusing the temporary rvalue and transferring ownership of its resources to the newly created `obj3`.

Let us now consider a final example:

```
void useObject(MyMovableClass obj)
{
    std::cout << "using object " << &obj << std::endl;
}

int main()
{
    MyMovableClass obj1(100); // constructor

    useObject(obj1);

    return 0;
}
```

In this case, an instance of `MyMovableClass`, `obj1`, is passed to a function `useObject` by value, thus making a copy of it.

Let us take an immediate look at the output of the program, before going into details:

```
(1)
CREATING instance of MyMovableClass at 0x7fffeefbff718 allocated with
size = 400 bytes
```

```
(2)
COPYING content of instance 0x7fffeefbff718 to instance 0x7fffeefbff708
```

```
using object 0x7fffeefbff708

(3)
DELETING instance of MyMovableClass at 0x7fffeefbff708

(4)
CREATING instance of MyMovableClass at 0x7fffeefbff6d8 allocated with
size = 800 bytes

(5)
MOVING (c'tor) instance 0x7fffeefbff6d8 to instance 0x7fffeefbff6e8

using object 0x7fffeefbff6e8

DELETING instance of MyMovableClass at 0x7fffeefbff6e8
DELETING instance of MyMovableClass at 0x7fffeefbff6d8
DELETING instance of MyMovableClass at 0x7fffeefbff718
```

First, we are creating an instance of `MyMovableClass`, `obj1`, by calling the constructor of the class (1).

Then, we are passing `obj1` by-value to a function `useObject`, which causes a temporary object `obj` to be instantiated, which is a copy of `obj1` (2) and is deleted immediately after the function scope is left (3).

Then, the function is called with a temporary instance of `MyMovableClass` as its

argument, which creates a temporary instance of `MyMovableClass` as an rvalue (4). But instead of making a copy of it as before, the move constructor is used (5) to transfer ownership of that temporary object to the function scope, which saves us one expensive deep-copy.



## Moving lvalues

There is one final aspect we need to look at: In some cases, it can make sense to treat lvalues like rvalues. At some point in your code, you might want to transfer ownership of a resource to another part of your program as it is not needed anymore in the current scope. But instead of copying it, you want to just move it as we have seen before. The "problem" with our implementation of `MyMovableClass` is that the call `useObject(obj1)` will trigger the copy constructor as we have seen in one of the last examples. But in order to move it, we would have to pretend to the compiler that `obj1` was an rvalue instead of an lvalue so that we can make an efficient move operation instead of an expensive copy.

There is a solution to this problem in C++, which is `std::move`. This function accepts an lvalue argument and returns it as an rvalue without triggering copy construction. So by passing an object to `std::move` we can force the compiler to use move semantics, either in the form of move constructor or the move assignment operator:

```
int main()
{
    MyMovableClass obj1(100); // constructor
```

```
useObject(std::move(obj1));  
  
    return 0;  
}
```

Nothing much has changed, apart from `obj1` being passed to the `std::move` function. The output would look like the following:

CREATING instance of MyMovableClass at 0x7fffeefbff718 allocated with  
size = 400 bytes

MOVING (c'tor) instance 0x7fffeefbff718 to instance 0x7fffeefbff708  
using object 0x7fffeefbff708

DELETING instance of MyMovableClass at 0x7fffeefbff708  
DELETING instance of MyMovableClass at 0x7fffeefbff718

By using `std::move`, we were able to pass the ownership of the resources within `obj1` to the function `useObject`. The local copy `obj1` in the argument list was created with the move constructor and thus accepted the ownership transfer from `obj1` to `obj`. Note that after the call to `useObject`, the instance `obj1` has been invalidated by setting its internal handle to null and thus may not be used anymore within the scope of `main` (even though you could theoretically try to access it, but this would be a really bad idea).



New and delete, they don't only allocate and deallocate memory,



they also call the constructor and the destructor of an object.



Guide

## Error-prone memory management with `new` and `delete`

In the previous chapters, we have seen that memory management on the heap using `malloc / free` or `new / delete` is extremely powerful, as they allow for a fine-grained control over the precious memory resource. However, the correct use of these concepts requires some degree of skill and experience (and concentration) from the programmer. If they are not handled correctly, bugs will quickly be introduced into the code. A major source of error is that the details around memory management with `new / delete` are completely left to the programmer. In the remainder of this lesson, the pair `malloc / free` will be omitted for reasons of brevity. However, many of the aspects that hold for `new / delete` will also apply to `malloc / free`.

Let us take a look at some of the worst problems with `new` and `delete`:

- 1. Proper pairing of new and delete :** Every dynamically allocated object that is created with `new` must be followed by a manual deallocation at a "proper" place in the program. If the programmer forgets to call `delete` (which can happen very quickly) or if it is done at an "inappropriate" position, memory leaks will occur which might clog up a large portion of memory.
- 2. Correct operator pairing :** C++ offers a variety of `new / delete` operators, especially when dealing with arrays on the heap. A dynamically allocated array initialized with `new[1]` may only be deleted with the operator `delete[1]`. If the wrong operator is

used, program behavior will be undefined - which is to be avoided at all cost in C++.

**3. Memory ownership :** If a third-party function returns a pointer to a data structure, the only way of knowing who will be responsible for resource deallocation is by looking into either the code or the documentation. If both are not available (as is often the case), there is no way to infer the ownership from the return type. As an example, in the final project of this course, we will use the graphical library wxWidgets to create the user interface of a chatbot application. In wxWidgets, the programmer can create child windows and control elements on the heap using `new`, but the framework will take care of deletion altogether. If for some reason the programmer does not know this, he or she might call `delete` and thus interfere with the inner workings of the wxWidgets library.

## The benefits of smart pointers

To put it briefly: Smart pointers were introduced in C++ to solve the above mentioned problems by providing a degree of automatic memory management: When a smart pointer is no longer needed (which is the case as soon as it goes out of scope), the memory to which it points is automatically deallocated. When contrasted with smart pointers, the conventional pointers we have seen so far are often termed "raw pointers".

In essence, smart pointers are classes that are wrapped around raw pointers. By overloading the `->` and `*` operators, smart pointer objects make sure that the memory to which their internal raw pointer refers to is properly deallocated. This makes it possible to use smart pointers with the same syntax as raw pointers. As soon as a smart pointer goes out of scope, its destructor is called and the block of memory to which the internal raw pointer refers is properly deallocated. This technique of wrapping a management class around a resource has been conceived by Bjarne Stroustrup and is called **Resource Acquisition Is Initialization (RAII)**. Before we continue with smart pointers and their usage let us take a close look at this powerful concept.

# Resource Acquisition Is Initialization

The RAII is a widespread programming paradigm, that can be used to protect a resource such as a file stream, a network connection or a block of memory which need proper management.

## Acquiring and releasing resources

In most programs of reasonable size, there will be many situations where a certain action at some point will necessitate a proper reaction at another point, such as:

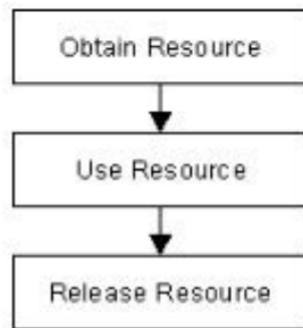
1. Allocating memory with `new` or `malloc`, which must be matched with a call to `delete` or `free`.
2. Opening a file or network connection, which must be closed again after the content has been read or written.
3. Protecting synchronization primitives such as atomic operations, memory barriers, monitors or critical sections, which must be released to allow other threads to obtain them.

The following table gives a brief overview of some resources and their respective allocation and deallocation calls in C++:

	Acquire	Release
Files	fopen	fclose
Memory	new, new[]	delete, delete[]
Locks	lock, try_lock	unlock

## The problem of reliable resource release

A general usage pattern common to the above examples looks like the following:



However, there are several problems with this seemingly simple pattern:

1. The program might throw an exception during resource use and thus the point of release might never be reached.
2. There might be several points where the resource could potentially be released, making it hard for a programmer to keep track of all eventualities.
3. We might simply forget to release the resource again.

## RAII to the rescue

The major idea of RAII revolves around object ownership and information hiding: Allocation and deallocation are hidden within the management class, so a programmer using the class does not have to worry about memory management responsibilities. If he has not directly allocated a resource, he will not need to directly deallocate it - whoever owns a resource deals with it. In the case of RAII this is the management class around the protected resource. The overall goal is to have allocation and deallocation (e.g. with `new` and `delete`) disappear from the surface level of the code you write.

RAII can be used to leverage - among others - the following advantages:

- Use class destructors to perform resource clean-up tasks such as proper memory deallocation when the RAII object gets out of scope
- Manage ownership and lifetime of dynamically allocated objects
- Implement encapsulation and information hiding due to resource acquisition and release being performed within the same object.

In the following, let us look at RAII from the perspective of memory management.

There are three major parts to an RAII class:

1. A resource is allocated in the constructor of the RAII class
2. The resource is deallocated in the destructor
3. All instances of the RAII class are allocated on the stack to reliably control the lifetime via the object scope

Let us now take a look at the code example on the right.

At the beginning of the program, an array of double values `den` is allocated on the stack. Within the loop, a new double is created on the heap using `new`. Then, the result of a division is printed to the console. At the end of the loop, `delete` is called to properly deallocate the heap memory to which `en` is pointing. Even though this code is working as it is supposed to, it is very easy to forget to call `delete` at the end. Let us therefore use the principles of RAII to create a management class that calls `delete` automatically:

```
class MyInt
{
    int *_p; // pointer to heap data
public:
    MyInt(int *p = NULL) { _p = p; }
    ~MyInt()
    {
        std::cout << "resource " << *_p << " deallocated" << std::endl;
        delete _p;
    }
    int &operator*() { return *_p; } // // overload dereferencing
                                    operator
};
```

In this example, the constructor of class `MyInt` takes a pointer to a memory resource. When the destructor of a `MyInt` object is called, the resource is deallocated from memory, which makes `MyInt` an RAII memory management class.

deleted from memory - which makes `MyInt` an RAII memory management class. Also, the `*` operator is overloaded which enables us to dereference `MyInt` objects in the same manner as with raw pointers. Let us therefore slightly alter our code example from above to see how we can properly use this new construct:

```
int main()
{
    double den[] = {1.0, 2.0, 3.0, 4.0, 5.0};
    for (size_t I = 0; I < 5; ++I)
    {
        // allocate the resource on the stack
        MyInt en(new int(i));

        // use the resource
        std::cout << *en << "/" << den[i] << " = " << *en / den[i] <<
        std::endl;
    }

    return 0;
}
```

Update the code on the right with the snippets above before proceeding.

Let us break down the resource allocation part in two steps:

1. The part `new int(i)` creates a new block of memory on the heap and initializes it with the value of `i`. The returned result is the address of the block of memory.
2. The part `MyInt en(...)` calls the constructor of class `MyInt`, passing the address of a valid memory block as a parameter.

After creating an object of class `MyInt` on the stack, which, internally, created an integer on the heap, we can use the dereference operator in the same manner as before to retrieve the value to which the internal raw pointer is pointing. Because the `MyInt` object `en` lives on the stack, it is automatically deallocated after each loop cycle - which automatically calls the destructor to release the heap memory. The following console output verifies this:

```
0/1 = 0
resource 0 deallocated
1/2 = 0.5
resource 1 deallocated
2/3 = 0.666667
resource 2 deallocated
3/4 = 0.75
resource 3 deallocated
4/5 = 0.8
resource 4 deallocated
```

We have thus successfully used the RAII idiom to create a memory management class that spares us from thinking about calling `delete`. By creating the `MyInt` object on the stack, we ensure that the deallocation occurs as soon as the object goes out of scope.

Quiz : What would be the major difference of the following program compared to the last example?

```
int main()
{
    double den[] = {1.0, 2.0, 3.0, 4.0, 5.0};
    for (size_t i = 0; i < 5; ++i)
    {
        // allocate the resource on the heap
        MyInt *en = new MyInt(new int(i));

        // use the resource
        std::cout << **en << "/" << den[i] << " = " << **en / den[i] <<
        std::endl;
    }

    return 0;
}
```

HIDE SOLUTION

ANSWER: The destructor of `MyInt` would never be called, hence causing a memory leak with each loop iteration.

## RAlI and smart pointers

In the last section, we have discussed the powerful RAlI idiom, which reduces the risk of improperly managed resources. Applied to the concept of memory management, RAlI enables us to encapsulate `new` and `delete` calls within a class and thus present the programmer with a clean interface to the resource he intends to use. Since C++11, there exists a language feature called *smart pointers*, which builds on the concept of RAlI and - without exaggeration - revolutionizes the way we use resources on the heap. Let's take a look.

## Smart pointer overview

Since C++11, the standard library includes *smart pointers*, which help to ensure that programs are free of memory leaks while also remaining exception-safe. With smart pointers, resource acquisition occurs at the same time that the object is initialized (when instantiated with `make_shared` or `make_unique`), so that all resources for the object are created and initialized in a single line of code.

In modern C++, raw pointers managed with `new` and `delete` should only be used in small blocks of code with limited scope, where performance is critical (such as with `placement new`) and ownership rights of the memory resource are clear. We will look at some guidelines on where to use which pointer later.

C++11 has introduced three types of smart pointers, which are defined in the header of the standard library:

1. The unique pointer `std::unique_ptr` is a smart pointer which exclusively owns a dynamically allocated resource on the heap. There must not be a second unique pointer to the same resource.

2. The shared pointer `std::shared_ptr` points to a heap resource but does not explicitly own it. There may even be several shared pointers to the same resource, each of which will increase an internal reference count. As soon as this count reaches zero, the resource will automatically be deallocated.
3. The weak pointer `std::weak_ptr` behaves similar to the shared pointer but does not increase the reference counter.

Prior to C++11, there was a concept called `std::auto_ptr`, which tried to realize a similar idea. However, this concept can now be safely considered as deprecated and should not be used anymore.

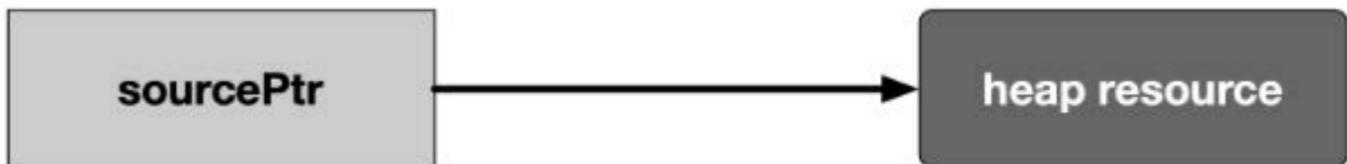
Let us now look at each of the three smart pointer types in detail.

## The Unique pointer

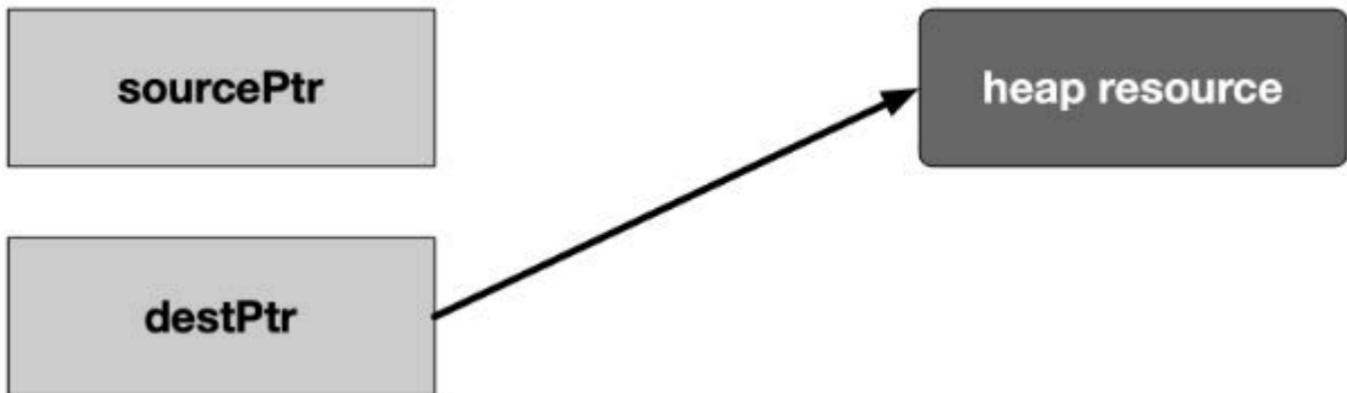
A unique pointer is the exclusive owner of the memory resource it represents. There must not be a second unique pointer to the same memory resource, otherwise there will be a compiler error. As soon as the unique pointer goes out of scope, the memory resource is deallocated again. Unique pointers are useful when working with a temporary heap resource that is no longer needed once it goes out of scope.

The following diagram illustrates the basic idea of a unique pointer:

```
auto sourcePtr = make_unique<MyObject>();
```



```
auto destPtr = std::move(sourcePtr);
```



In the example, a resource in memory is referenced by a unique pointer instance `sourcePtr`. Then, the resource is reassigned to another unique pointer instance `destPtr` using `std::move`. The resource is now owned by `destPtr` while `sourcePtr` can still be used but does not manage a resource anymore.

A unique pointer is constructed using the following syntax:

```
std::unique_ptr<Type> p(new Type);
```

In the example on the right we will see how a unique pointer is constructed and how it compares to a raw pointer.

The function `RawPointer` contains the familiar steps of (1) allocating memory on the heap with `new` and storing the address in a pointer variable, (2) assigning a value to the memory block using the dereferencing operator `*` and (3) finally deleting the resource on the heap. As we already know, forgetting to call `delete` will result in a memory leak.

The function `UniquePointer` shows how to achieve the same goal using a smart pointer from the standard library. As can be seen, a smart pointer is a class template that is declared on the stack and then initialized by a raw pointer (returned by `new`) to a heap-allocated object. The smart pointer is now responsible for deleting the memory that the raw pointer specifies - which happens as soon as the smart pointer goes out of scope. Note that smart pointers always need to be declared on the stack, otherwise the scoping mechanism would not work.

The smart pointer destructor contains the call to `delete`, and because the smart pointer is declared on the stack, its destructor is invoked when the smart pointer goes out of scope, even if an exception is thrown.

In the example now on the right, we will construct a unique pointer to a custom class. Also, we will see how the standard `->` and `*` operators can be used to access member functions of the managed object, just as we would with a raw pointer:

Note that the custom class `MyClass` has two constructors, one without arguments and one with a string to be passed, which initializes a member variable `_text` that lives on the stack. Also, once an object of this class gets destroyed, a message to the console is printed, along with the value of `_text`. In `main`, two unique pointers are created with the address of a `MyClass` object on the heap as arguments. With `myClass2`, we can see that constructor arguments can be passed just as we would with raw pointers. After both pointers have been created, we can use the `->` operator to access members of the class, such as calling the function `setText`. From looking at the function call alone you would not be able to tell that `myClass1` is in fact a smart pointer. Also, we can use the dereference operator `*` to access the value of `myClass1` and `myClass2` and assign the one to the other. Finally, the `.` operator gives us access to proprietary functions of the smart pointer, such as retrieving the internal raw pointer with `get()`.

The console output of the program looks like the following:

```
Objects have stack addresses 0x1004000e0 and 0x100400100
String 2 destroyed
String 2 destroyed
```

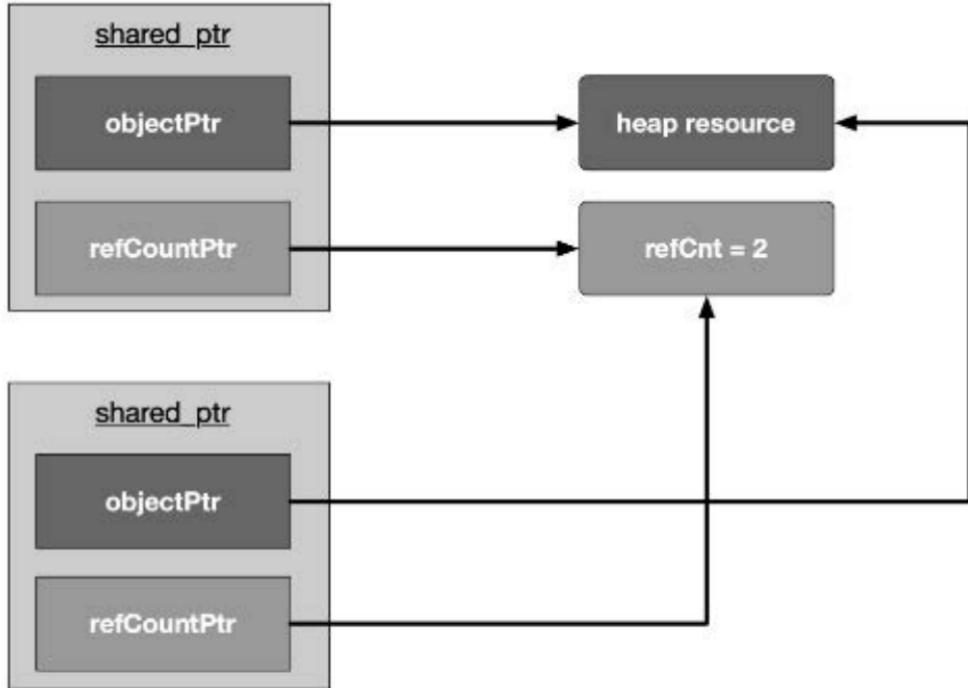
Obviously, both pointers have different addresses on the stack, even after copying the contents from `myClass2` to `myClass1`. As can be seen from the last two lines of the output, the destructor of both objects gets called automatically at the end of the program and - as expected - the value of the internal string is identical due to the copy operation.

Summing up, the unique pointer allows a single owner of the underlying internal raw pointer. Unique pointers should be the default choice unless you know for certain that sharing is required at a later stage. We have already seen how to transfer ownership of a resource using the Rule of Five and move semantics. Internally, the unique pointer uses this very concept along with RAII to encapsulate a resource (the raw pointer) and transfer it between pointer objects when either the move assignment operator or the move constructor are called. Also, a key feature of a unique pointer, which makes it so well-suited as a return type for many functions, is the possibility to convert it to a shared pointer. We will have a deeper look into this in the section on ownership transfer.

## The Shared Pointer

Just as the unique pointer, a shared pointer owns the resource it points to. The main difference between the two smart pointers is that shared pointers keep a reference counter on how many of them point to the same memory resource. Each time a shared pointer goes out of scope, the counter is decreased. When it reaches zero (i.e. when the last shared pointer to the resource is about to vanish), the memory is properly deallocated. This smart pointer type is useful for cases where you require access to a memory location on the heap in multiple parts of your program and you want to make sure that whoever owns a shared pointer to the memory can rely on the fact that it will be accessible throughout the lifetime of that pointer.

The following diagram illustrates the basic idea of a shared pointer:



Please take a look at the code on the right.

We can see that shared pointers are constructed just as unique pointers are. Also, we can access the internal reference count by using the method `use_count()`. In the inner block, a second shared pointer `shared2` is created and `shared1` is assigned to it. In the copy constructor, the internal resource pointer is copied to `shared2` and the resource counter is incremented in both `shared1` and `shared2`. Let us take a look at the output of the code:

```
shared pointer count = 1
shared pointer count = 2
shared pointer count = 1
```



You may have noticed that the lifetime of `shared2` is limited to the scope denoted by the enclosing curly brackets. Thus, once this scope is left and `shared2` is destroyed, the reference counter in `shared1` is decremented by one - which is reflected in the three console outputs given above.

A shared pointer can also be redirected by using the `reset()` function. If the resource which a shared pointer manages is no longer needed in the current scope, the pointer can be reset to manage a difference resource as illustrated in the example on the right.

Note that in the example, the destructor of `MyClass` prints a string to the console when called. The output of the program looks like the following:

```
shared pointer count = 1
Destructor of MyClass called
shared pointer count = 1
Destructor of MyClass called
```

After creation, the program prints `1` as the reference count of `shared`. Then, the `reset` function is called with a new instance of `MyClass` as an argument. This causes the destructor of the first `MyClass` instance to be called, hence the console output. As can be seen, the reference count of the shared pointer is still at `1`. Then, at the end of the program, the destructor of the second `MyClass` object is called once the path of execution leaves the scope of `main`.

Despite all the advantages of shared pointers, it is still possible to have problems with memory management though. Consider the scenario on the right.

In main, two shared pointers `myClass1` and `myClass2` which are managing objects of type `MyClass` are allocated on the stack. As can be seen from the console output, both smart pointers are automatically deallocated when the scope of main ends:

```
Destructor of MyClass called  
Destructor of MyClass called
```

When the following two lines are added to main, the result is quite different:

```
myClass1->_member = myClass2;  
myClass2->_member = myClass1;
```

These two lines produce a *circular reference*. When `myClass1` goes out of scope at the end of main, its destructor can't clean up memory as there is still a reference count of 1 in the smart pointer, which is caused by the shared pointer `_member` in `myClass2`. The same holds true for `myClass2`, which can not be properly deleted as there is still a shared pointer to it in `myClass1`. This deadlock situation prevents the destructors from being called and causes a memory leak. When we use *Valgrind* on this program, we get the following summary:

```
--20360-- LEAK SUMMARY:  
--20360--   definitely lost: 16 bytes in 1 blocks  
--20360--   indirectly lost: 80 bytes in 3 blocks  
--20360--   possibly lost: 72 bytes in 3 blocks  
--20360--   still reachable: 200 bytes in 6 blocks  
--20360--   suppressed: 18,985 bytes in 160 blocks
```

As can be seen, the memory leak is clearly visible with 16 bytes being marked as "definitely lost". To prevent such circular references, there is a third smart pointer, which we will look at in the following.

## The Weak Pointer

Similar to shared pointers, there can be multiple weak pointers to the same resource. The main difference though is that weak pointers do not increase the reference count. Weak pointers hold a non-owning reference to an object that is managed by another shared pointer.

The following rule applies to weak pointers: You can only create weak pointers out of shared pointers or out of another weak pointer. The code on the right shows a few examples of how to use and how not to use weak pointers.

The output looks as follows:

```
shared pointer count = 1  
shared pointer count = 1
```

First, a shared pointer to an integer is created with a reference count of 1 after creation. Then, two weak pointers to the integer resource are created, the first directly from the shared pointer and the second indirectly from the first weak pointer. As can be seen from the output, neither of both weak pointers increased the reference count. At the end of main, the attempt to directly create a weak pointer to an integer resource would lead to a compile error.

As we have seen with raw pointers, you can never be sure whether the memory resource to which the pointer refers is still valid. With a weak pointer, even though this type does not prevent an object from being deleted, the validity of its resource can be checked. The code on the right illustrates how to use the `expired()` function to do this.

Thus, with smart pointers, there will always be a managing instance which is responsible for the proper allocation and deallocation of a resource. In some cases it might be necessary to convert from one smart pointer type to another. Let us take a look at the set of possible conversions in the following.

## Converting between smart pointers

The example on the right illustrates how to convert between the different pointer types.

In (1), a conversion from **unique pointer** to **shared pointer** is performed. You can see that this can be achieved by using `std::move`, which calls the move assignment operator on `sharedPtr1` and steals the resource from `uniquePtr` while at the same time invalidating its resource handle on the heap-allocated integer.

In (2), you can see how to convert from **weak** to **shared** pointer. Imagine that you have been passed a weak pointer to a memory object which you want to work on. To avoid invalid memory access, you want to make sure that the object will not be deallocated before your work on it has been finished. To do this, you can convert a weak pointer to a shared pointer by calling the `lock()` function on the weak pointer.

In (3), a **raw pointer** is extracted from a shared pointer. However, this operation does not decrease the reference count within `sharedPtr2`. This means that calling `delete` on `rawPtr` in the last line before main returns will generate a runtime error as a resource is trying to be deleted which is managed by `sharedPtr2` and has already been removed. The output of the program when compiled with `g++` thus is:

```
malloc: *** error for object 0x1003001f0: pointer being freed was not allocated
```

Note that there are no options for converting away from a shared pointer. Once you have created a shared pointer, you must stick to it (or a copy of it) for the remainder of your program.

## When to use raw pointers and smart pointers?

As a general rule of thumb with modern C++, smart pointers should be used often. They will make your code safer as you no longer need to think (much) about the proper allocation and deallocation of memory. As a consequence, there will be much fewer memory leaks caused by dangling pointers or crashes from accessing invalidated memory blocks.

When using raw pointers on the other hand, your code might be susceptible to the following bugs:

1. Memory leaks
2. Freeing memory that shouldn't be freed
3. Freeing memory incorrectly
4. Using memory that has not yet been allocated
5. Thinking that memory is still allocated after being freed

With all the advantages of smart pointers in modern C++, one could easily assume that it would be best to completely ban the use of new and delete from your code. However, while this is in many cases possible, it is not always advisable as well. Let us take a look at the [C++ core guidelines](#), which has several **rules for explicit memory allocation and deallocation**. In the scope of this course, we will briefly discuss three of them:

1. R. 10: Avoid malloc and free While the calls

`(MyClass*)malloc( sizeof(MyClass) )` and `new MyClass` both allocate a block of memory on the heap in a perfectly valid manner, only `new` will also call the constructor of the class and `free` the destructor. To reduce the risk of undefined behavior, `malloc` and `free` should thus be avoided.

2. R. 11: Avoid calling `new` and `delete` explicitly Programmers have to make sure that every call of `new` is paired with the appropriate `delete` at the correct position so that no memory leak or invalid memory access occur. The emphasis here lies in the word "explicitly" as opposed to implicitly, such as with smart pointers or containers in the standard template library.

3. R. 12: Immediately give the result of an explicit resource allocation to a manager object It is recommended to make use of manager objects for controlling resources such as files, memory or network connections to mitigate the risk of memory leaks. This is the core idea of smart pointers as discussed at length in this section.

Summarizing, raw pointers created with `new` and `delete` allow for a high degree of flexibility and control over the managed memory as we have seen in earlier lessons of this course. To mitigate their proneness to errors, the following additional recommendations can be given:

- A call to `new` should not be located too far away from the corresponding `delete`. It is bad style to stretch your `new / delete` pairs throughout your program with references criss-crossing your entire code.
- Calls to `new` and `delete` should always be hidden from third parties so that they must not concern themselves with managing memory manually (which is similar to R. 12).

In addition to the above recommendations, the C++ core guidelines also contain a total of 13 rules for the [recommended use of smart pointers](#). In the following, we will discuss a selection of these:

1. R. 20 : Use `unique_ptr` or `shared_ptr` to represent ownership
2. R. 21 : Prefer `unique_ptr` over `std::shared_ptr` unless you need to share ownership

Both pointer types express ownership and responsibilities (R. 20). A `unique_ptr` is an exclusive owner of the managed resource; therefore, it cannot be copied, only moved. In contrast, a `shared_ptr` shares the managed resource with others. As described above, this mechanism works by incrementing and decrementing a common reference counter. The resulting administration overhead makes `shared_ptr` more expensive than `unique_ptr`. For this reason `unique_ptr` should always be the first choice (R. 21).

4. R. 22 : Use `make_shared()` to make `shared_ptr`
5. R. 23 : Use `make_unique()` to make `std::unique_ptr`

The increased management overhead compared to raw pointers becomes in particular true if a `shared_ptr` is used. Creating a `shared_ptr` requires (1) the allocation of the resource using `new` and (2) the allocation and management of the reference counter. Using the factory function `make_shared` is a one-step operation with lower overhead and should thus always be preferred. (R.22). This also holds for `unique_ptr` (R.23), although the performance gain in this case is minimal (if existent)

at all).

But there is an additional reason for using the `make_...` factory functions: Creating a smart pointer in a single step removes the risk of a memory leak. Imagine a scenario where an exception happens in the constructor of the resource. In such a case, the object would not be handled properly and its destructor would never be called - even if the managing object goes out of scope. Therefore, `make_shared` and `make_unique` should always be preferred. Note that `make_unique` is only available with compilers that support at least the C++14 standard.

### 3. R. 24 : Use `weak_ptr` to break cycles of `shared_ptr`

We have seen that weak pointers provide a way to break a deadlock caused by two owning references which are cyclicly referring to each other. With weak pointers, a resource can be safely deallocated as the reference counter is not increased.

The remaining set of guideline rules referring to smart pointers are mostly concerning the question of how to pass a smart pointer to a function. We will discuss this question in the next concept.



## Passing smart pointers to functions

Let us consider the following recommendation of the C++ guidelines on smart pointers:

R. 30 : Take smart pointers as parameters only to explicitly express lifetime semantics

The core idea behind this rule is the notion that functions that only manipulate objects without affecting its lifetime in any way should not be concerned with a particular kind of smart pointer. A function that does not manipulate the lifetime or ownership should use raw pointers or references instead. A function should take smart pointers as parameter only if it examines or manipulates the smart pointer itself. As we have seen, smart pointers are classes that provide several features such as counting the references of a `shared_ptr` or increasing them by making a copy. Also, data can be moved from one `unique_ptr` to another and thus transferring the ownership. A particular function should accept smart pointers only if it expects to do something of this sort. If a function just needs to operate on the underlying object without the need of using any smart pointer property, it should accept the objects via raw pointers or references instead.

The following examples are pass-by-value types that lend the ownership of the underlying object:

1. `void f(std::unique_ptr<MyObject> ptr)`
2. `void f(std::shared_ptr<MyObject> ptr)`
3. `void f(std::weak_ptr<MyObject> ptr)`

Passing smart pointers by value means to lend their ownership to a particular function `f`. In the above examples 1-3, all pointers are passed by value, i.e. the function `f` has a private copy of it which it can (and should) modify. Depending on the type of smart pointer, a tailored strategy needs to be used. Before going into details, let us take a look at the underlying rule from the C++ guidelines (where "widget" can be understood as "class").

The basic idea of a `unique_ptr` is that there exists only a single instance of it. This is why it can't be copied to a local function but needs to be moved instead with the function `std::move`. The code example on the right illustrates the principle of transferring the object managed by the unique pointer `uniquePtr` into a function `f`.

The class `MyClass` has a private object `_member` and a public function `printVal()` which prints the address of the managed object (`this`) as well as the member value to the console. In `main`, an instance of `MyClass` is created by the factory function `make_unique()` and assigned to a unique pointer instance `uniquePtr` for management. Then, the pointer instance is moved into the function `f` using move semantics. As we have not overloaded the move constructor or move assignment operator in `MyClass`, the compiler is using the default implementation. In `f`, the address of the copied / moved unique pointer `ptr` is printed and the function `printVal()` is called on it. When the path of execution returns to `main()`, the program checks for the validity of `uniquePtr` and, if valid, calls the function `printVal()` on it again. Here is the console output of the program:



```
unique_ptr 0x7fffeefbff710, managed object 0x100300060 with val = 23
```

```
unique_ptr 0x7fffeefbff6f0, managed object 0x100300060 with val = 23
```

The output nicely illustrates the copy / move operation. Note that the address of `unique_ptr` differs between the two calls while the address of the managed object as well as of the value are identical. This is consistent with the inner workings of the move constructor, which we overloaded in a previous section. The copy-by-value behavior of `f()` creates a new instance of the unique pointer but then switches the address of the managed `MyClass` instance from source to destination. After the move is complete, we can still use the variable `uniquePtr` in `main` but it now is only an empty shell which does not contain an object to manage.

When passing a shared pointer by value, move semantics are not needed. As with unique pointers, there is an underlying rule for transferring the ownership of a shared pointer to a function:

#### R.34: Take a `shared_ptr` parameter to express that a function is part owner

Consider the example on the right. The main difference in this example is that the `MyClass` instance is managed by a shared pointer. After creation in `main()`, the address of the pointer object as well as the current reference count are printed to the console. Then, `sharedPtr` is passed to the function `f()` by value, i.e. a copy is made. After returning to `main`, pointer address and reference counter are printed again. Here is what the output of the program looks like:

```
shared_ptr (ref_cnt= 1) 0x7fffeefbff708, managed object 0x100300208 with val = 23
shared_ptr (ref_cnt= 2) 0x7fffeefbff6e0, managed object 0x100300208 with val = 23
shared_ptr (ref_cnt= 1) 0x7fffeefbff708, managed object 0x100300208 with val = 23
```

Throughout the program, the address of the managed object does not change. When passed to `f()`, the reference count changes to 2. After the function returns and the local `shared_ptr` is destroyed, the reference count changes back to 1. In summary, move semantics are usually not needed when using shared pointers. Shared pointers can be passed by value safely and the main thing to remember is that with each pass, the internal reference counter is increased while the managed object stays the same.

Without giving an example here, the `weak_ptr` can be passed by value as well, just like the shared pointer. The only difference is that the pass does not increase the reference counter.

With the above examples, pass-by-value has been used to lend the ownership of smart pointers. Now let us consider the following additional rules from the C++ guidelines on smart pointers:

R.33: Take a `unique_ptr&` parameter to express that a function reseats the widget and

R.35: Take a `shared_ptr&` parameter to express that a function might reseat the shared pointer

Both rules recommend passing-by-reference, when the function is supposed to modify the ownership of an existing smart pointer and not a copy. We pass a non-const reference to a `unique_ptr` to a function if it might modify it in any way, including deletion and reassignment to a different resource.

Passing a `unique_ptr` as `const` is not useful as the function will not be able to do anything with it: Unique pointers are all about proprietary ownership and as soon as the pointer is passed, the function will assume ownership. But without the right to modify the pointer, the options are very limited.

A `shared_ptr` can either be passed as const or non-const reference. The const should be used when you want to express that the function will only read from the pointer or it might create a local copy and share ownership.

Lastly, we will take a look at **passing raw pointers** and references. The general rule of thumb is that we can use a simple raw pointer (which can be null) or a plain reference (which can ~~not~~ be null), when the function we are passing will only inspect the managed object without modifying the smart pointer. The internal (raw) pointer to the object can be retrieved using the `get()` member function. Also, by providing access to the raw pointer, you can use the smart pointer to manage memory in your own code and pass the raw pointer to code that does not support smart pointers.

When using raw pointers retrieved from the `get()` function, you should take special care not to delete them or to create new smart pointers from them. If you did so, the ownership rules applying to the resource would be severely violated. When passing a raw pointer to a function or when returning it (see next section), raw pointers should always be considered as owned by the smart pointer from which the raw reference to the resource has been obtained.

## Returning smart pointers from functions

With return values, the same logic that we have used for passing smart pointers to functions applies: Return a smart pointer, both unique or shared, if the caller needs to manipulate or access the pointer properties. In case the caller just needs the underlying object, a raw pointer should be returned.

Smart pointers should always be returned by value. This is not only simpler but also has the following advantages:

1. The overhead usually associated with return-by-value due to the expensive copying process is significantly mitigated by the built-in move semantics of smart pointers. They only hold a reference to the managed object, which is quickly switched from destination to source during the move process.
2. Since C++17, the compiler used *Return Value Optimization* (RVO) to avoid the copy usually associated with return-by-value. This technique, together with *copy-elision*, is able to optimize even move semantics and smart pointers (not in call cases though, they are still an essential part of modern C++).

3. When returning a *shared\_ptr* by value, the internal reference counter is guaranteed to be properly incremented. This is not the case when returning by pointer or by reference.

The topic of smart pointers is a complex one. In this course, we have covered many basics and some of the more advanced concepts. However, there are many more aspects to consider and features to use when integrating smart pointers into your code. The full set of smart pointer rules in the C++ guidelines is a good start to dig deeper into one of the most powerful features of modern C++.

In the previous section, we have taken a look at the three smart pointer types in C++. In addition to smart pointers, you are now also familiar with move semantics, which is of particular importance in this section. In the following, we will discuss how to properly pass and return smart pointers to functions and vice-versa. In modern C++, there are various ways of doing this and in many cases, the method of choice has an impact on both performance and code robustness. The basis of this section are the C++ core guidelines on smart pointers, some of which we will be examining in the following.

## Best-Practices for Passing Smart Pointers

This section contains a condensed summary of when (and when not) to use smart pointers and how to properly pass them between functions. This section is intended as a guide for your future use of this important feature in modern C++ and will hopefully encourage you not to ditch raw pointers altogether but instead to think about where your code could benefit from smart pointers - and when it would most probably not.

The following list contains all the variations (omitting `const`) of passing an object to a function:

```
void f( object* ); // (a)
void f( object& ); // (b)
void f( unique_ptr<object> ); // (c)
void f( unique_ptr<object>& ); // (d)
void f( shared_ptr<object> ); // (e)
void f( shared_ptr<object>& ); // (f)
```

### The Preferred Way

The preferred way of to pass object parameters is by using a) or b):

```
void f( object* );
void f( object& );
```

In doing so, we do not have to worry about the lifetime policy a caller might have implemented. Using a specific smart pointer in a case where we only want to observe an object or manipulate a member might be overly restrictive.

With the non-owning raw pointer `*` or the reference `&` we can observe an object from which we can assume that its lifetime will exceed the lifetime of the function parameter. In concurrency however, this might not be the case, but for linear code we can safely assume this.

To decide whether a `*` or `&` is more appropriate, you should think about whether you need to express that there is no object. This can only be done with pointers by passing e.g. `nullptr`. In most other cases, you should use a reference instead.

## The Object Sink

The preferred way of passing an object to a function so that the function takes ownership of the object (or „consumes“ it) is by using method c) from the above list:

```
void f( unique_ptr<object> );
```

In this case, we are passing a unique pointer by value from caller to function, which then takes ownership of the the pointer and the underlying object. This is only possible using move semantics as there may be only a single reference to the object managed by the unique pointer.

After the object has been passed in this way, the caller will have an invalid unique pointer and the function to which the object now belongs may destroy it or move it somewhere else.

Using `const` with this particular call does not make sense as it models an ownership transfer so the source will be definitely modified.

## In And Out Again 1

In some cases, we want to modify a unique pointer (not necessarily the underlying object) and re-use it in the context of the caller. In this case, method d) from the above list might be most suitable:

```
void f( unique_ptr<object>& );
```

Using this call structure, the function states that it might modify the smart pointer, e.g. by redirecting it to another object. It is not recommended to use it for accepting an object only because we should avoid restricting ourselves unnecessarily to a particular object lifetime strategy on the caller side.

Using `const` with this call structure is not recommendable as we would not be able to modify the `unique_ptr` in this case. In case you want to modify the underlying object, use method a) instead.

## Sharing Object Ownership

In the last examples, we have looked at strategies involving unique ownership. In this example, we want to express that a function will store and share ownership of an object on the heap. This can be achieved by using method e) from the list above:

```
void f( shared_ptr<object> )
```

In this example, we are making a copy of the shared pointer passed to the function. In doing so, the internal reference counter within all shared pointers referring to the same heap object is incremented by one.

This strategy can be recommended for cases where the function needs to retain a copy of the shared\_ptr and thus share ownership of the object. This is of interest when we need access to smart pointer functions such as the reference count or we must make sure that the object to which the shared pointer refers is not prematurely deallocated (which might happen in concurrent programming).

If the local scope of the function is not the final destination, a shared pointer can also be moved, which does not increase the reference count and is thus more effective.

A disadvantage of using a shared\_ptr as a function argument is that the function will be limited to using only objects that are managed by shared pointers - which limits flexibility and reusability of the code.

## In And Out Again 2

As with unique pointers, the need to modify shared pointers and re-use them in the context of the caller might arise. In this case, method f) might be the right choice:

```
void f( shared_ptr<object>& );
```

This particular way of passing a shared pointer expresses that the function f will modify the pointer itself. As with method e), we will be limiting the usability of the function to cases where the object is managed by a shared\_ptr and nothing else.

## Last Words

The topic of smart pointers is a complex one. In this course, we have covered many basics and some of the more advanced concepts. However, for some cases there are more aspects to consider and features to use when integrating smart pointers into your code. The full set of smart pointer rules in the C++ guidelines is a good start to dig deeper into one of the most powerful features of modern C++.

# Memory Management in C++ **Final Project**

Overview of Tasks

So before we dive into the code,

# Final Project

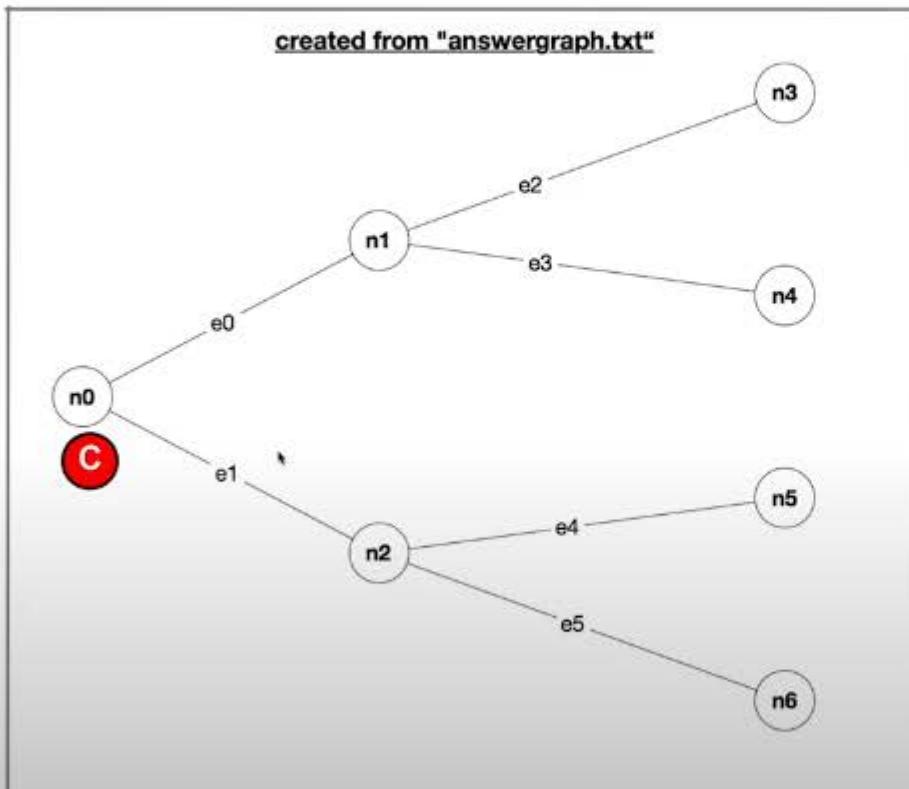
## Overview

- **Purpose :** To analyze an existing ChatBot program, which is able to discuss some memory management topics based on the content of a knowledge base. The program can be executed and works as intended. However, no advanced concepts as discussed in this course have been used. There are no smart pointers, no move semantics and not much thought has been given on ownership and on memory allocation.
- **Your task :** Use the course knowledge to optimize the ChatBot program from a memory management perspective. There is a total of five tasks to be completed.

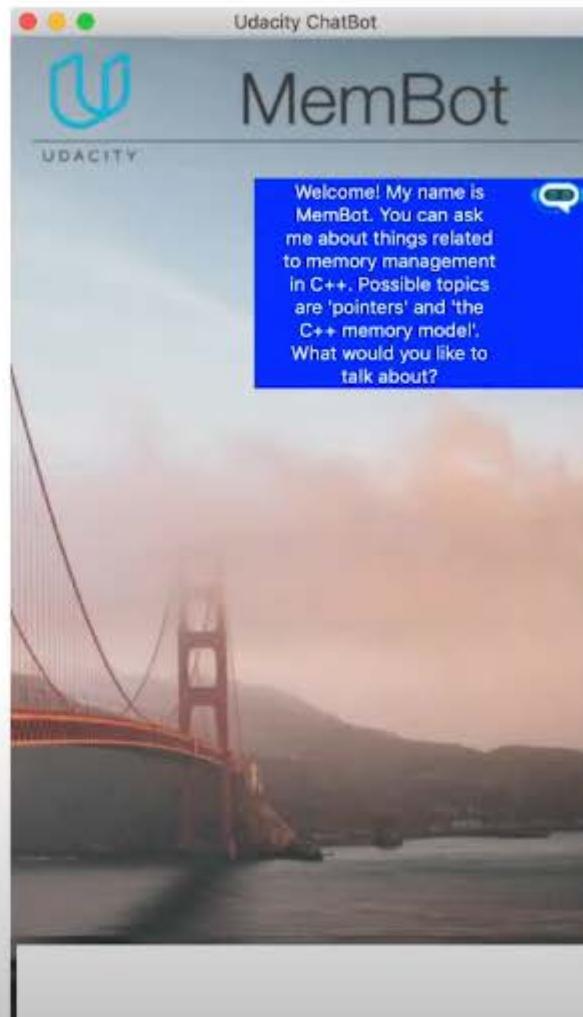
In total, we have a number of five tasks which you have to complete.

# Final Project

## Demo

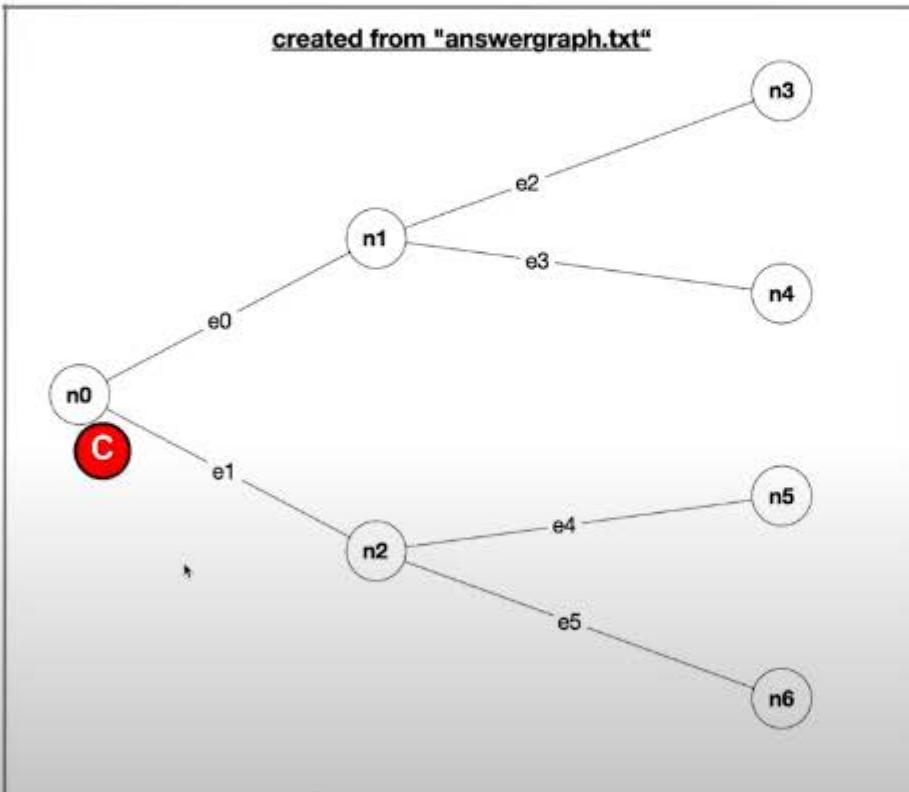


E0 corresponds to pointers and e1 corresponds to memory model.

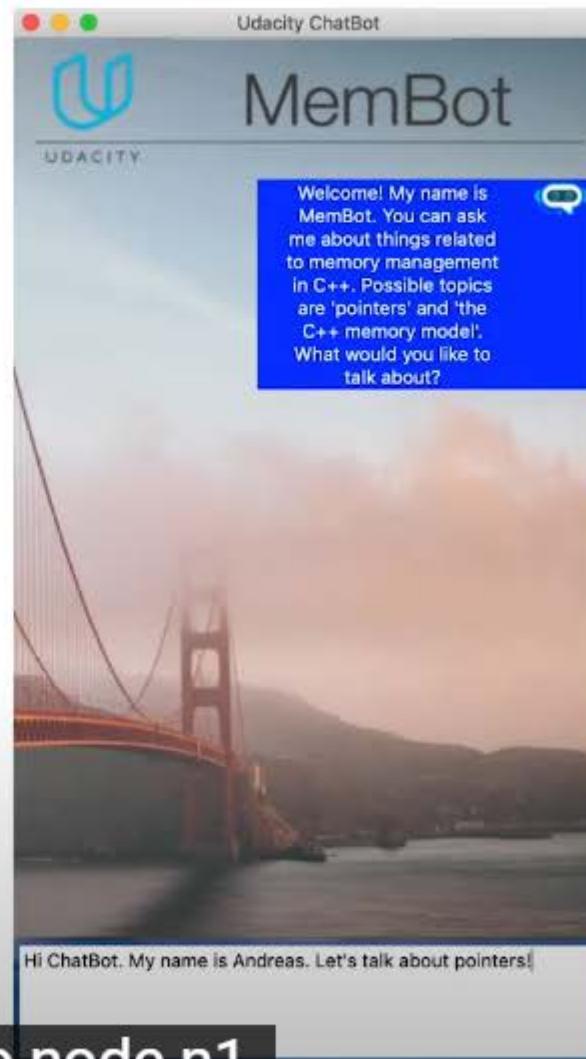


# Final Project

## Demo

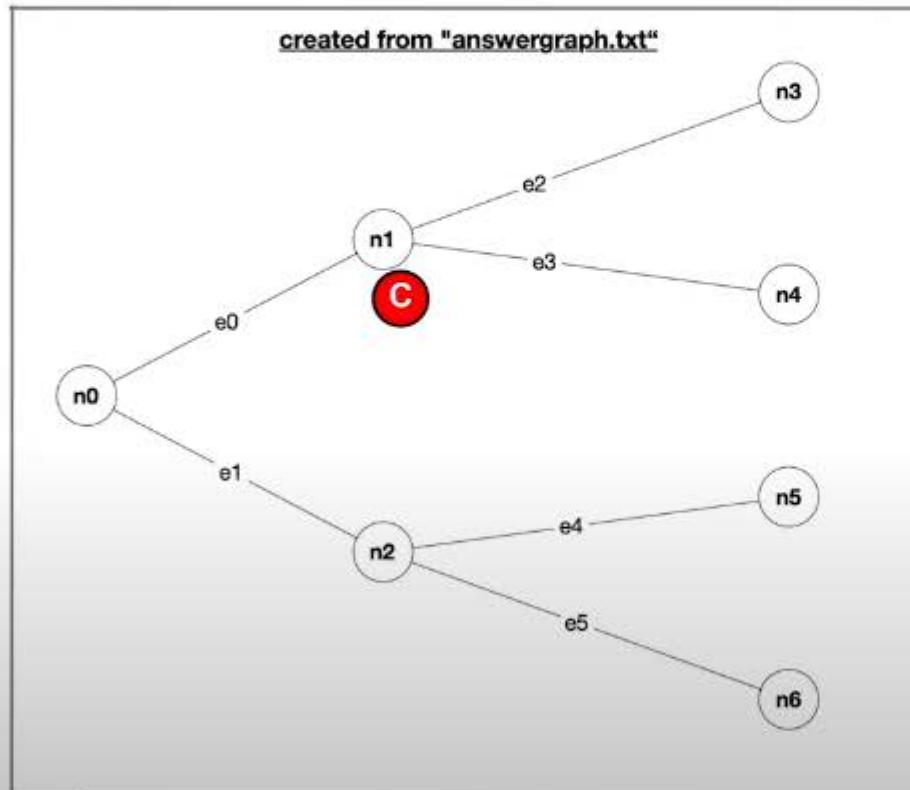


All the chatbot is moving to node n1,

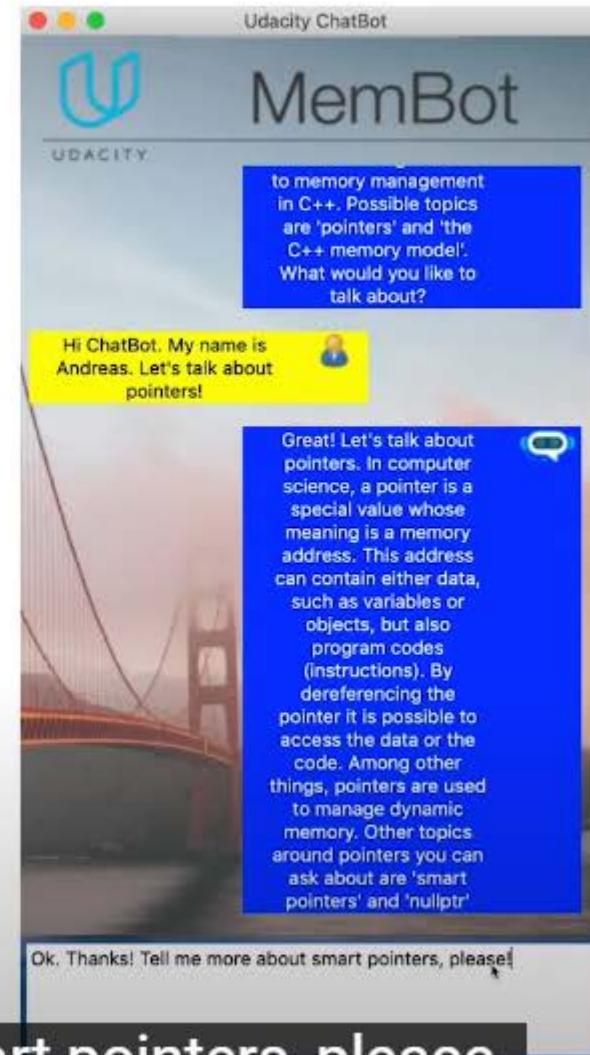


# Final Project

## Demo

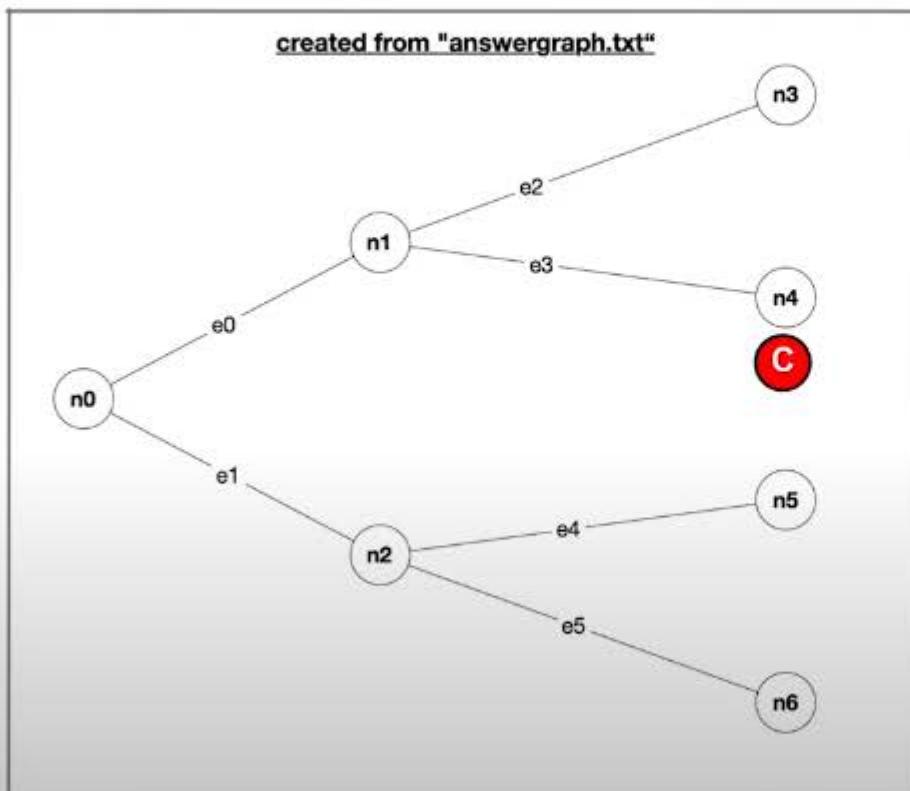


and I want to know more about smart pointers, please.

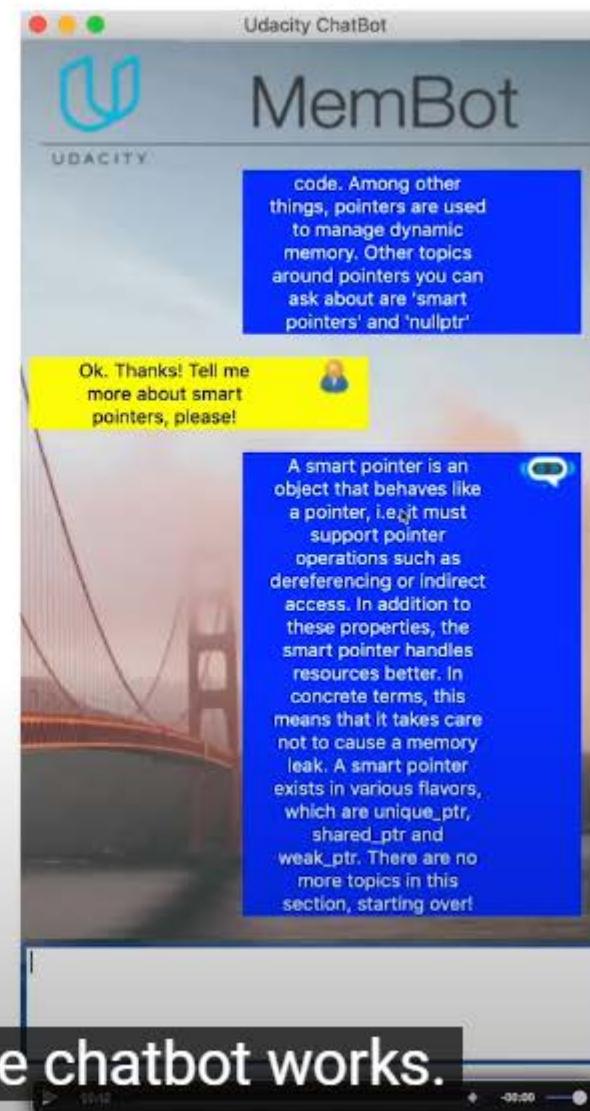


# Final Project

## Demo



So that's the basic idea of how the chatbot works.



## Project Introduction Summary

**Purpose** : In this project you will analyze and modify an existing ChatBot program, which is able to discuss some memory management topics based on the content of a knowledge base. The program can be executed and works as intended. However, no advanced concepts as discussed in this course have been used. There are no smart pointers, no move semantics and not much thought has been given on ownership and on memory allocation.

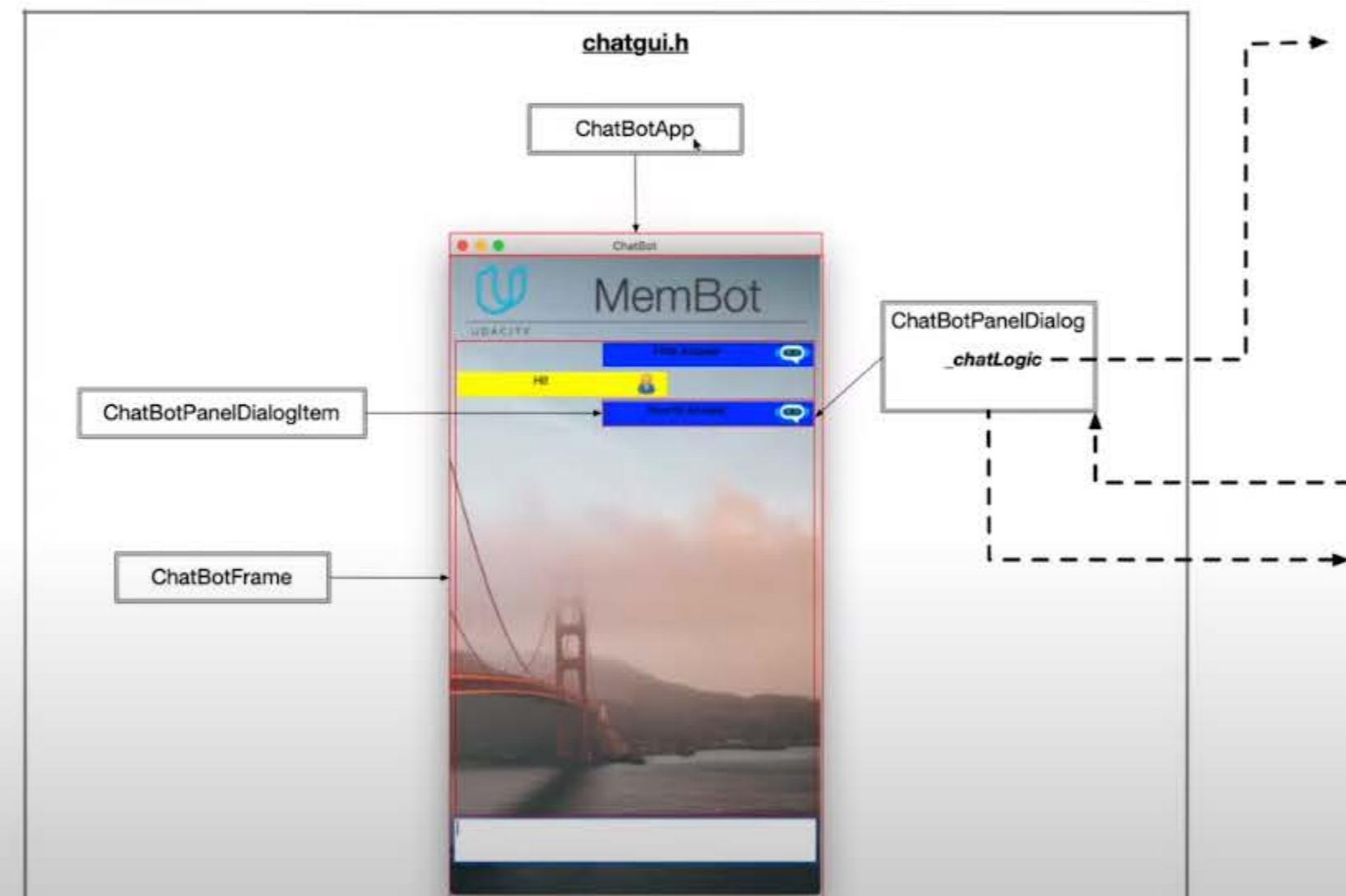
**Your task** : Use the course knowledge to optimize the ChatBot program from a memory management perspective. There is a total of five tasks to be completed.

You can find the GitHub repo for the project [here](#). The rubric for the project can be found [here](#).



# Final Project

## Program schematic



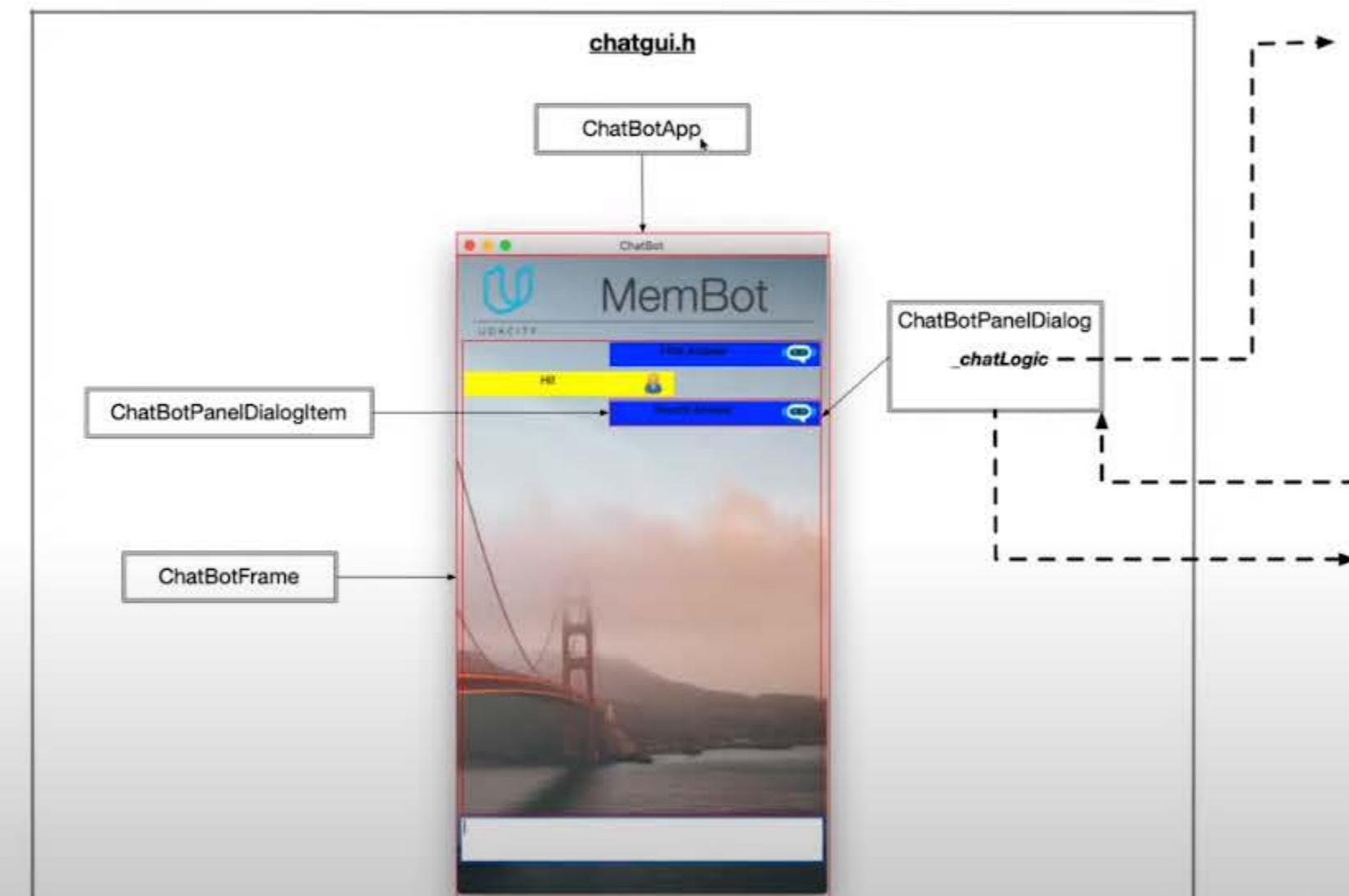
So WX widgets is one among many different ways of creating





# Final Project

## Program schematic



a graphical user interface for your C++ program



0:40 / 2:24



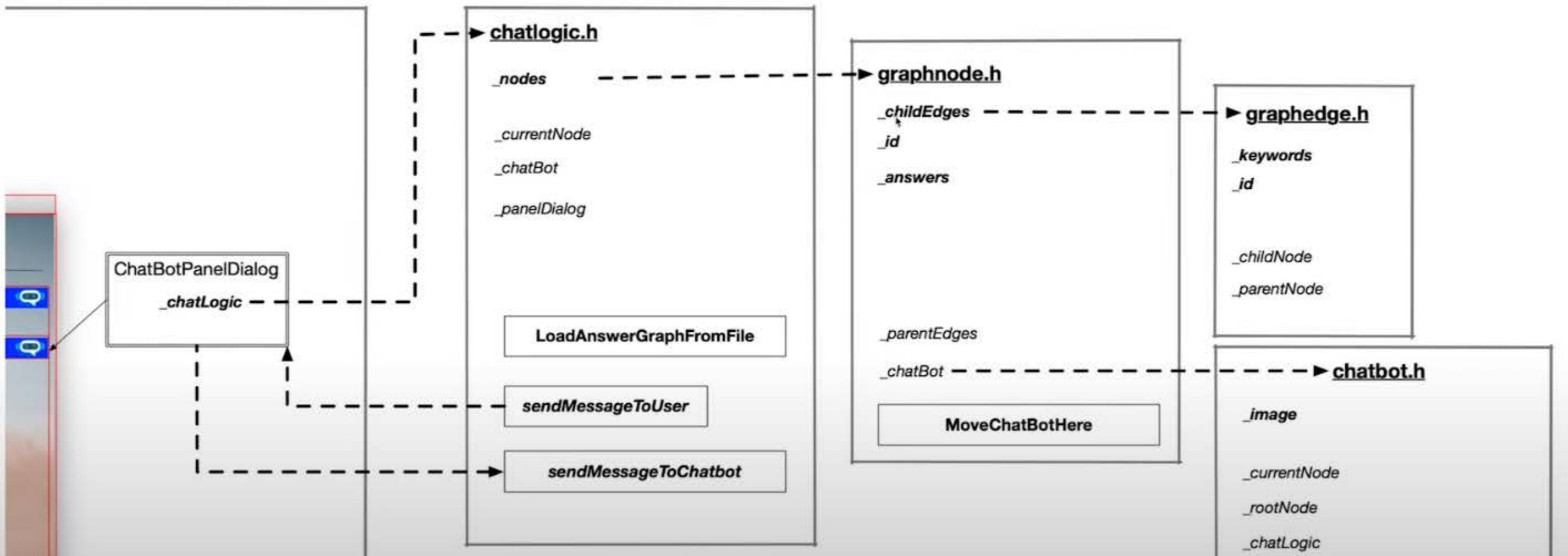
YouTube





# Final Project

## Program schematic



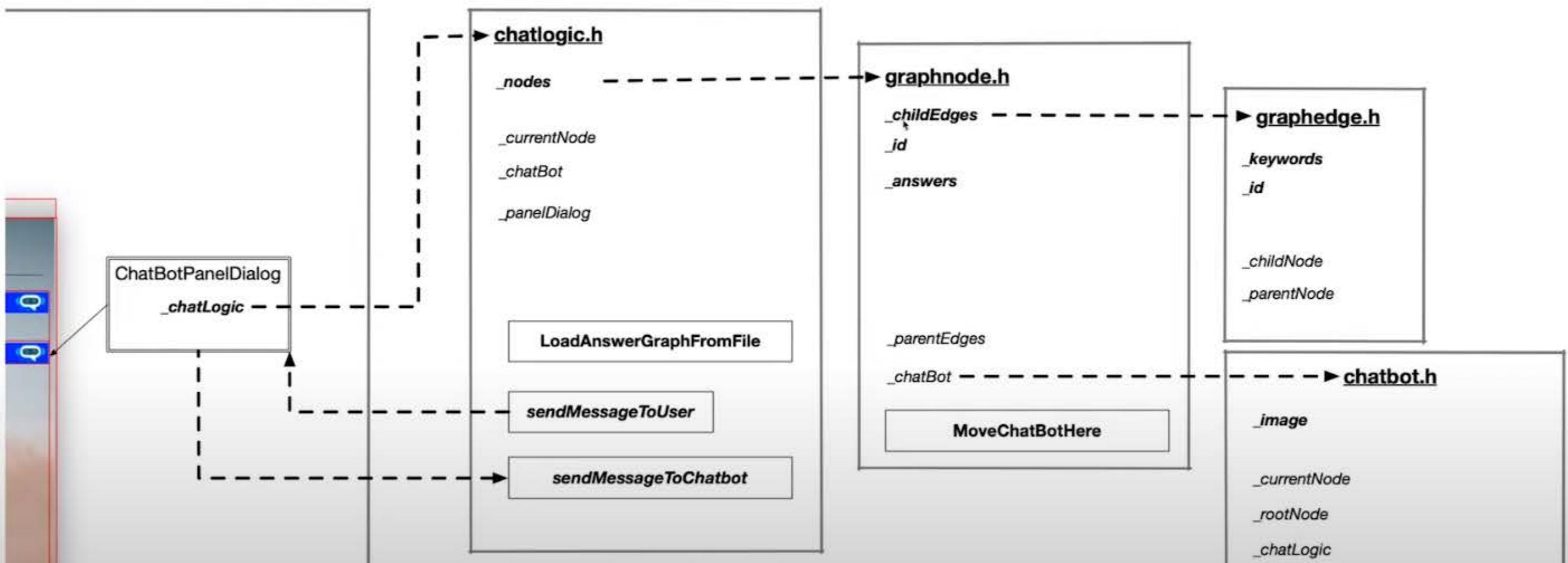
In a node, we have a number of edges which are the edges leading to child nodes.





# Final Project

## Program schematic



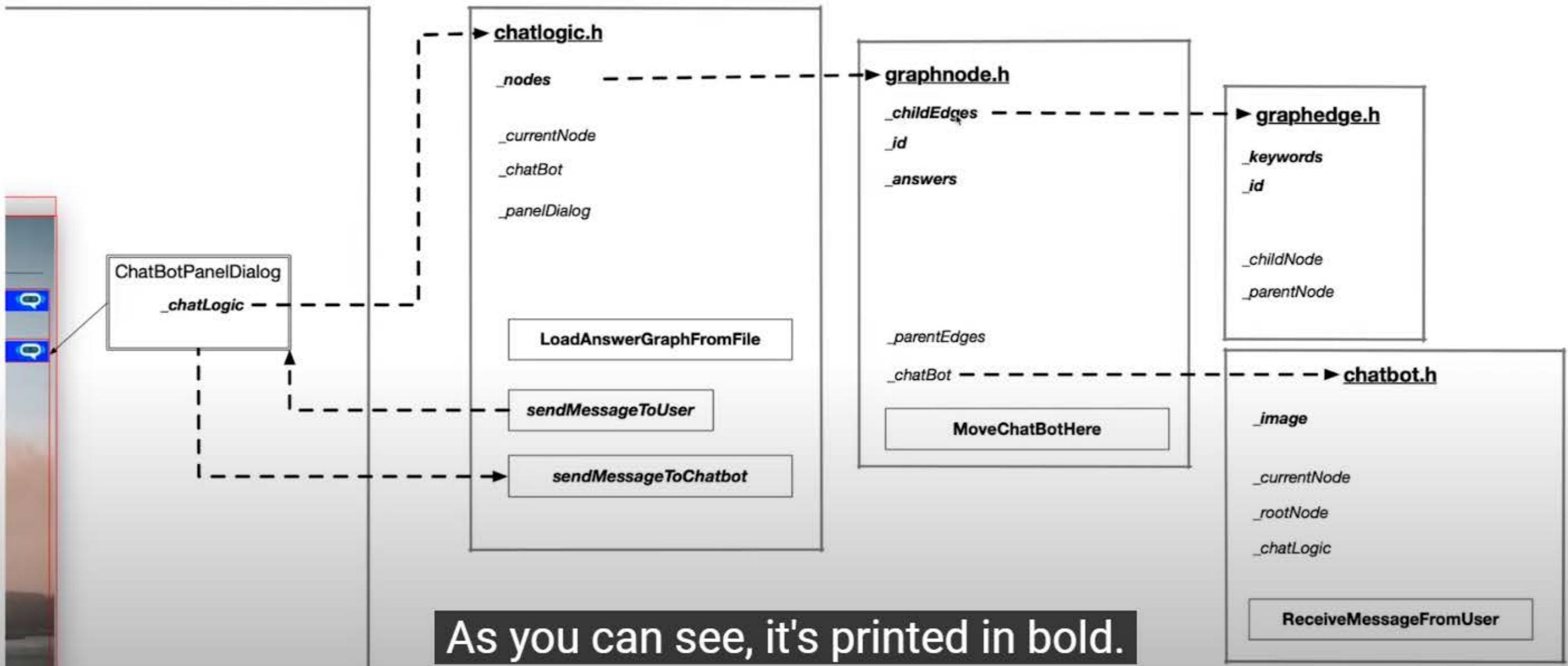
So all the nodes which come after a certain note has been reached for example,





# Final Project

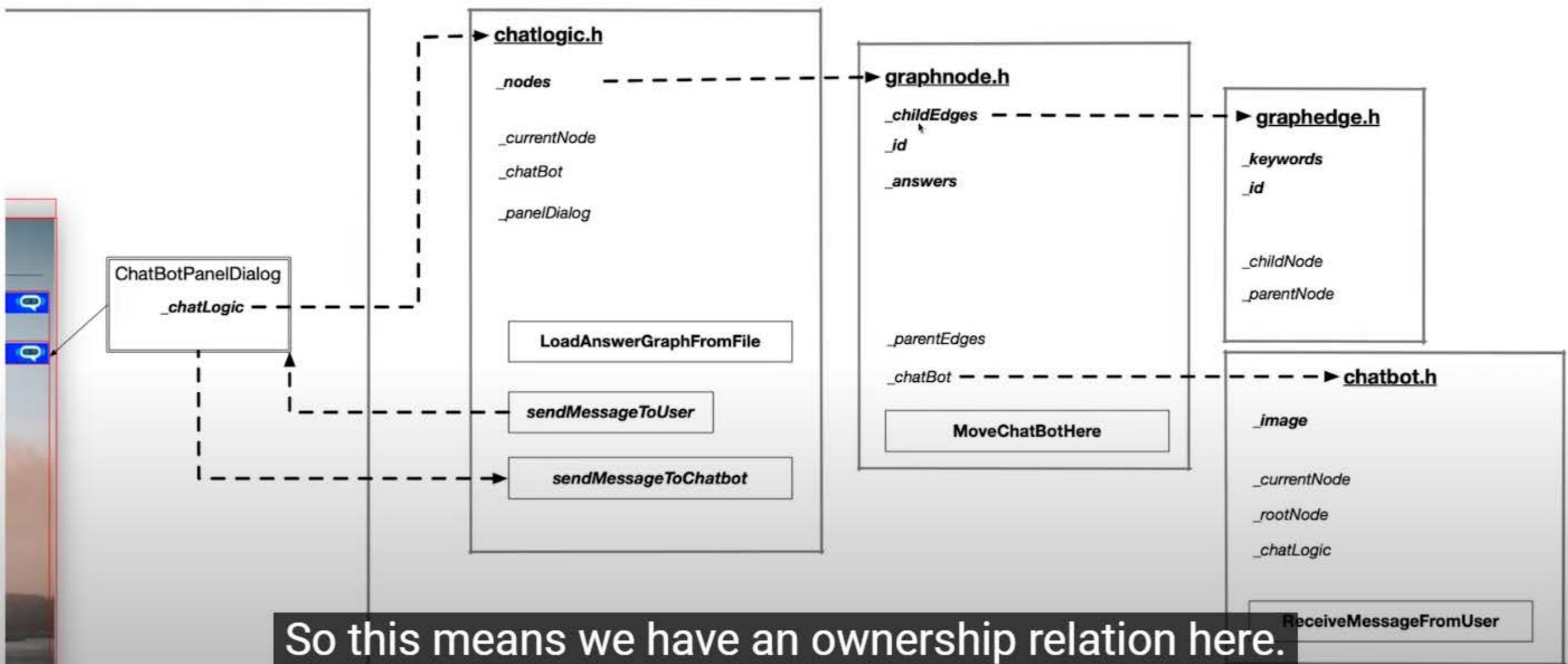
## Program schematic





# Final Project

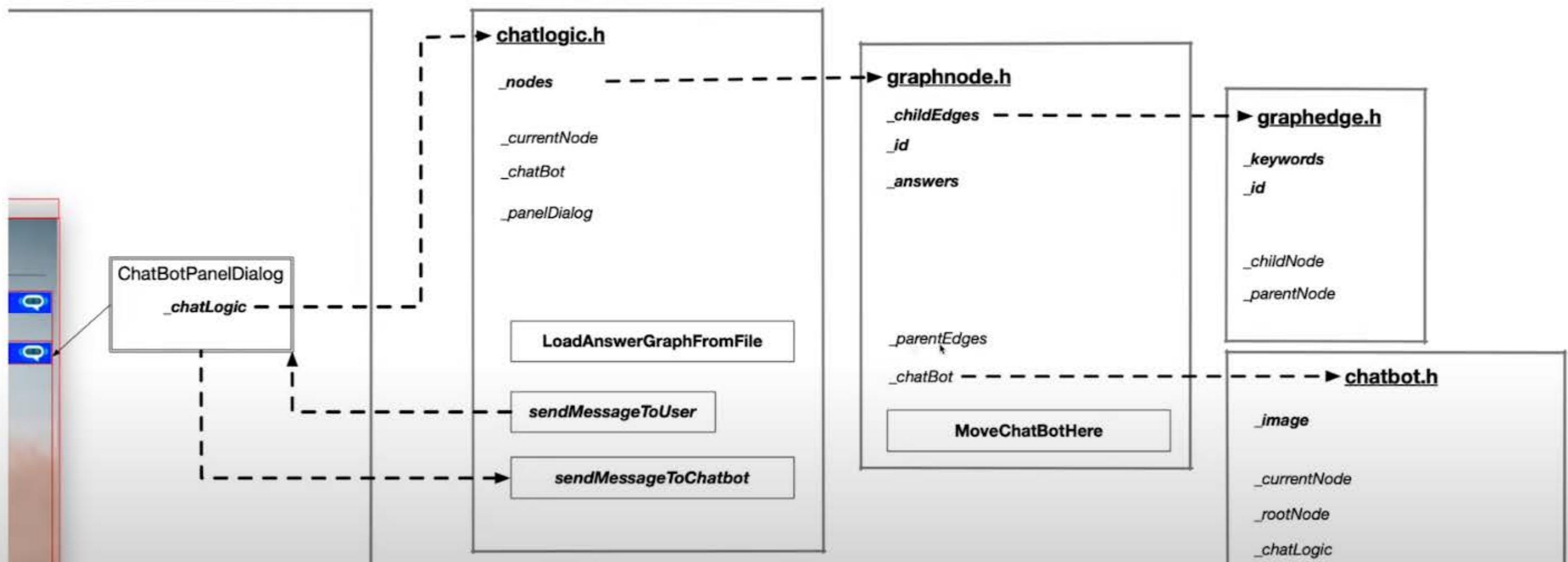
## Program schematic





# Final Project

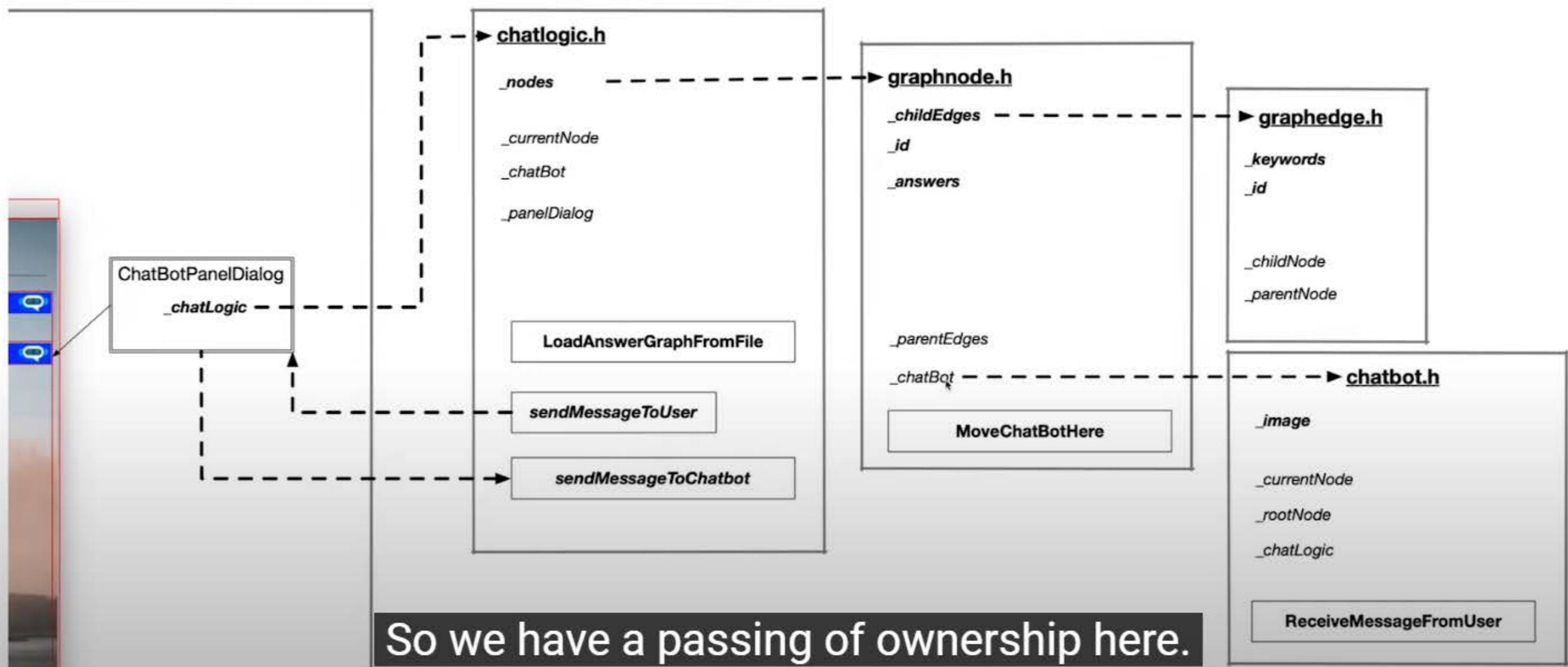
## Program schematic



We have handles for parent edges but node does not own the edges which led to it.

# Final Project

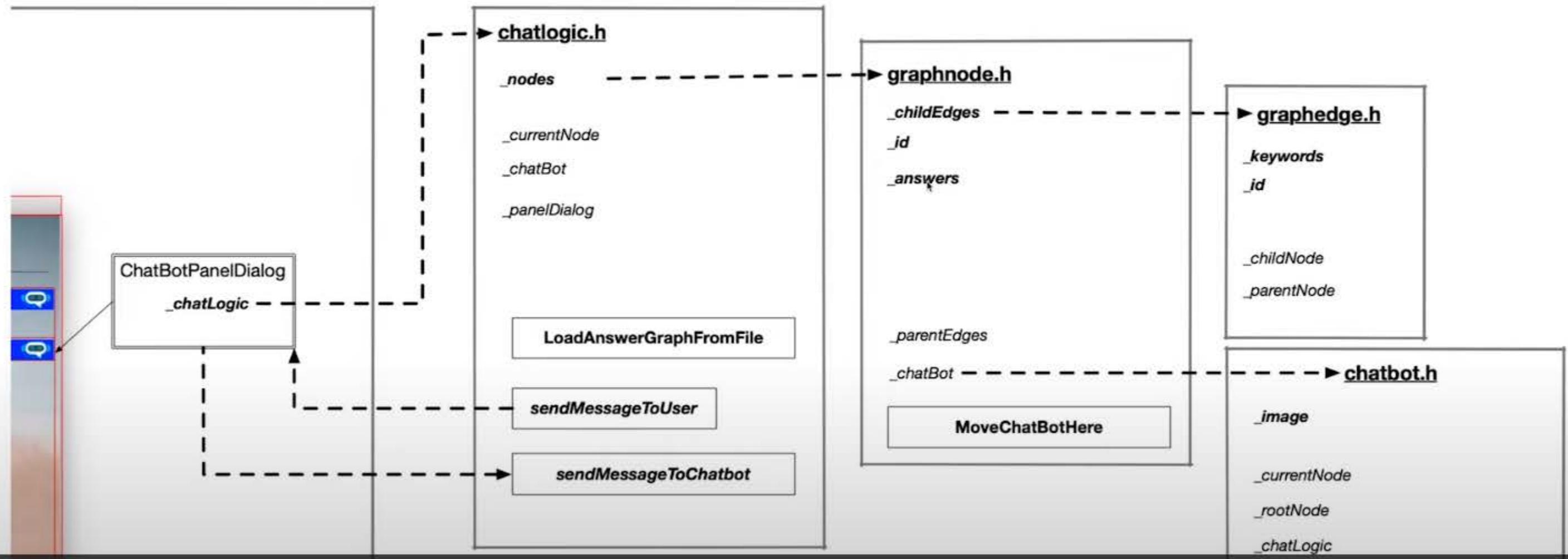
## Program schematic





# Final Project

## Program schematic



also the answers which the chat bot can give are contained here in this variable answer.





# Final Project

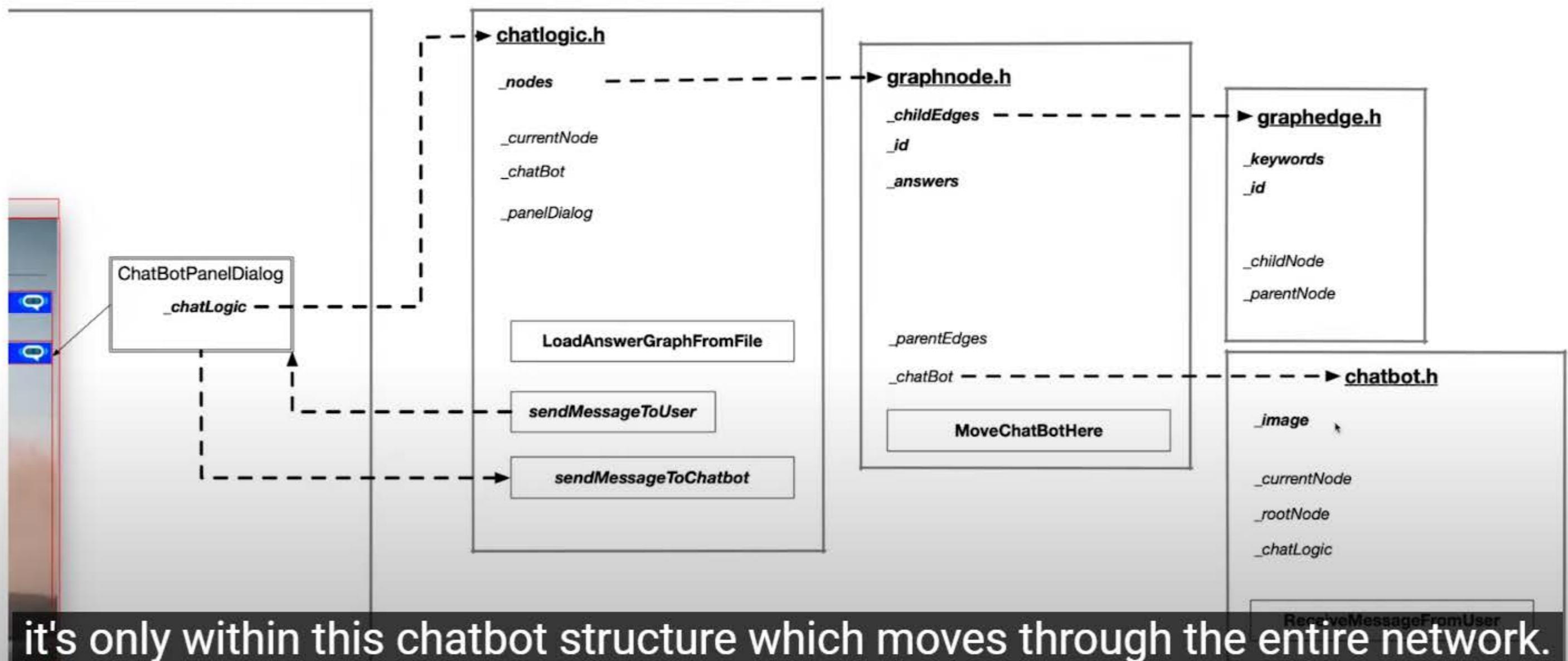
## Program schematic





# Final Project

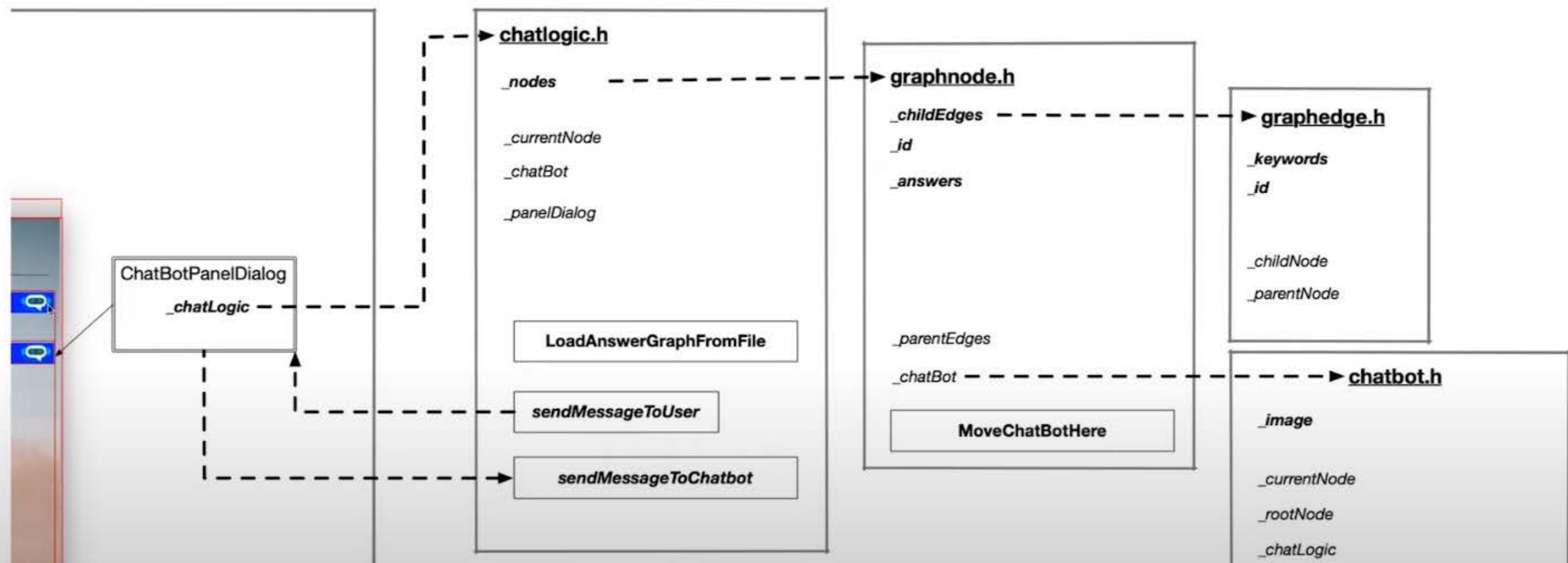
## Program schematic





# Final Project

## Program schematic

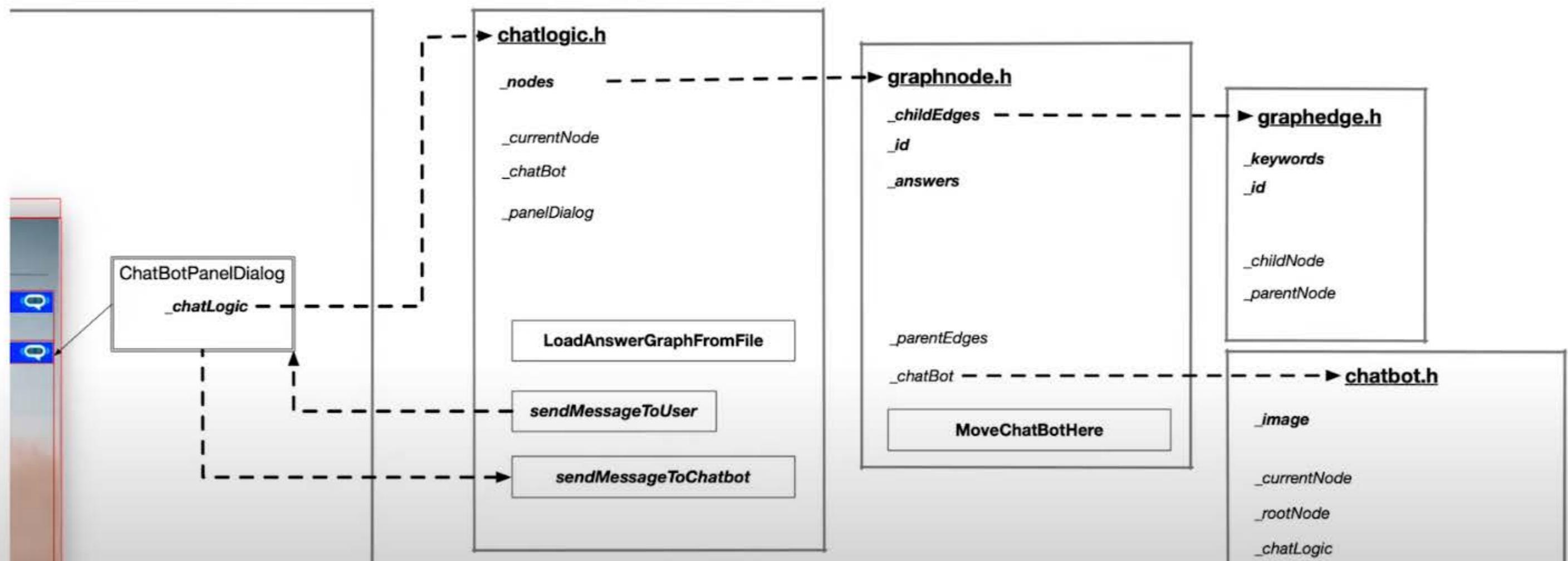


each new answer despite the chatbot having been moved from one node to the next.



# Final Project

## Program schematic

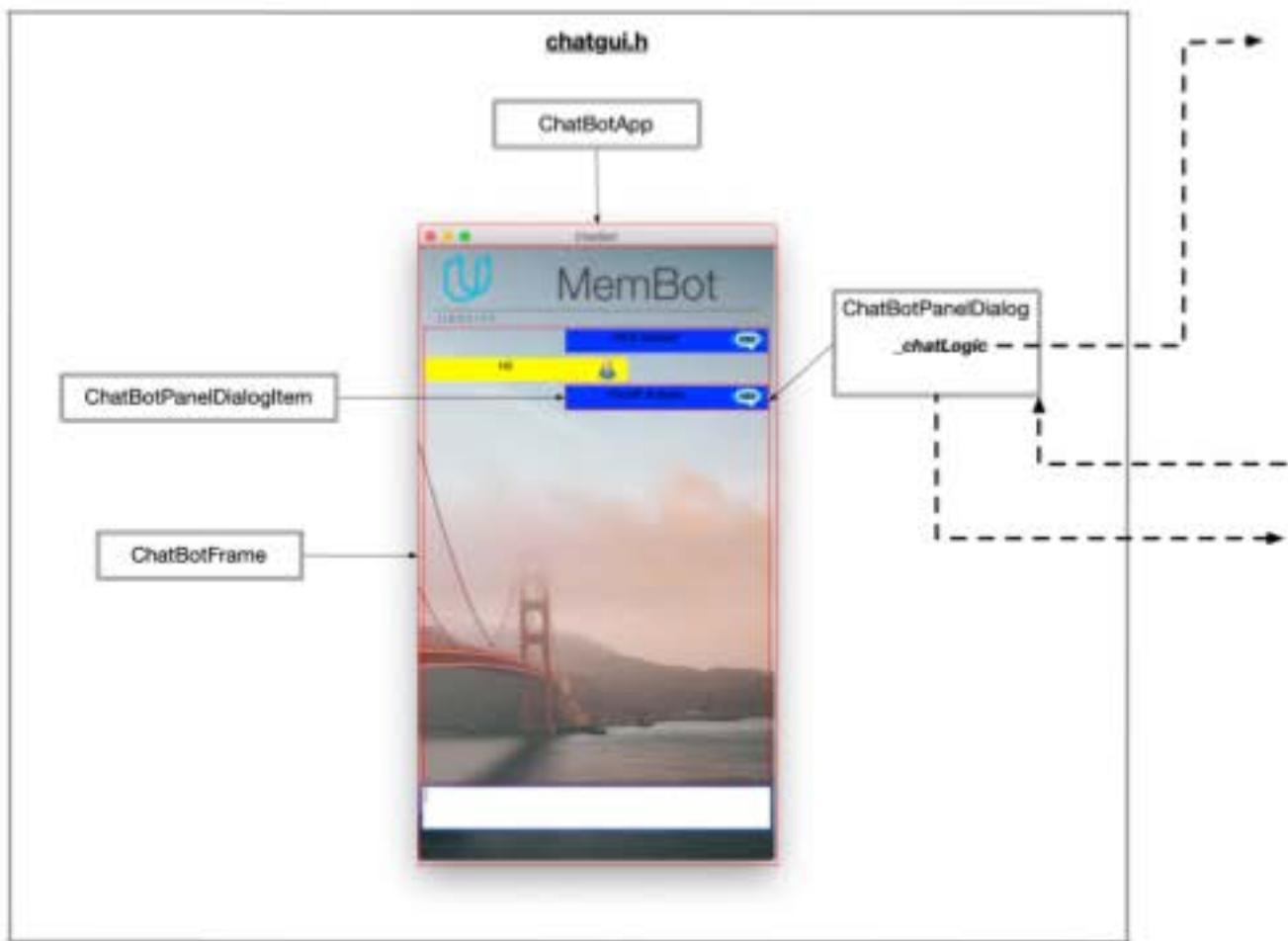


each new answer despite the chatbot having been moved from one node to the next.



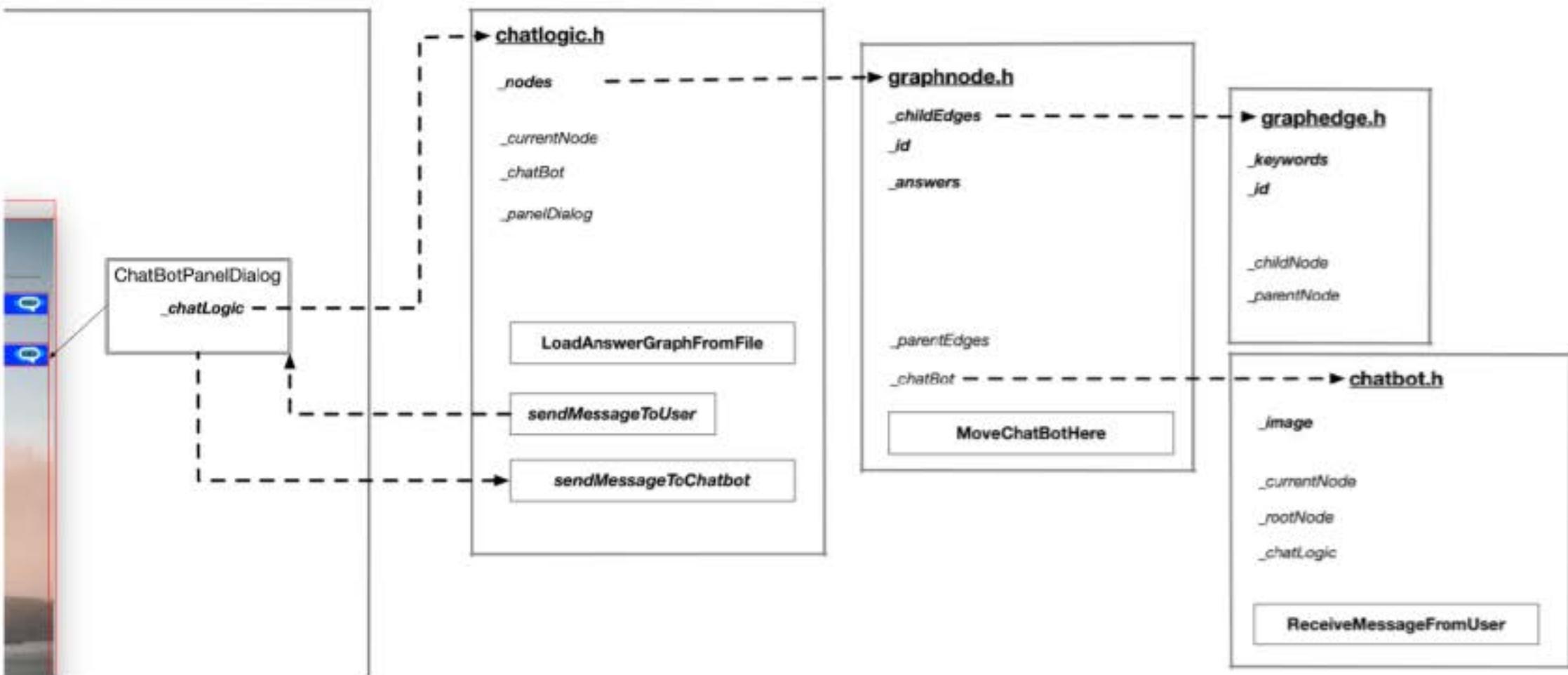
# Final Project

## Program schematic



# Final Project

## Program schematic



# Final Project

## Knowledge Base

// define all graph nodes

<TYPE:NODE><ID:0><ANSWER>Welcome! My name is MemBot. You can ask me about things related to memory management in C++. Possible topics are 'pointers' and 'the C++ memory model'. What would you like to talk about?>

<TYPE:NODE><ID:1><ANSWER>Great! Let's talk about pointers. In computer science, a pointer is a special value whose meaning is a memory address. This address can contain either data, such as variables or objects, but also program codes (instructions). By dereferencing the pointer it is possible to access the data or the code. Among other things, pointers are used to manage dynamic memory. Other topics around pointers you can ask about are 'smart pointers' and 'nullptr'>

<TYPE:NODE><ID:2><ANSWER>When instantiating variables, programmers can choose whether to do this on the heap, on the stack or in static memory. Do you want to know more about those two concepts? Simply ask me about 'heap', 'stack' or 'static'.>

...

// connect nodes with edges

<TYPE:EDGE><ID:0><PARENT:0><CHILD:1><KEYWORD:pointer><KEYWORD:smart pointer>

<TYPE:EDGE><ID:1><PARENT:0><CHILD:2><KEYWORD:memory model><KEYWORD:heap><KEYWORD:stack>

...

Let's have a look now at the knowledge base.





# Final Project

## Your Tasks

- **Task 5 : Moving the ChatBot**

In file `chatlogic.cpp`, create a local `ChatBot` instance on the stack at the bottom of function `LoadAnswerGraphFromFile`. Then, use move semantics to pass the `ChatBot` instance into the root node. Make sure that `ChatLogic` has no ownership relation to the `ChatBot` instance and thus is no longer responsible for memory allocation and deallocation.

Note that the member `\_chatBot` remains so it can be used as a communication handle between GUI and `ChatBot` instance. Make all required changes in files `chatlogic.h` / `chatlogic.cpp` and `graphnode.h` / `graphnode.cpp`. When the program is executed, messages on which part of the Rule of Five components of `ChatBot` is called should be printed to the console.

So it's an intentional bug,





# Final Project

## Your Tasks

- **Task 1 : Exclusive Ownership 1**

In file `chatgui.h` / `chatgui.cpp`, make `_chatLogic` an exclusive resource to class ``ChatbotPanelDialog`` using an appropriate smart pointer. Where required, make changes to the code such that data structures and function parameters reflect the new structure.

- **Task 2 : The Rule Of Five**

In file `chatbot.h` / `chatbot.cpp`, make changes to the class `ChatBot` such that it complies with the Rule of Five. Make sure to properly allocate / deallocate memory resources on the heap and also copy member data where it makes sense to you. In each of the methods (e.g. the copy constructor), print a string of the type „`ChatBot Copy Constructor`” to the console so that you can see which method is called in later examples.

You have five tasks in total.





# Final Project

## Your Tasks

- **Task 3 : Exclusive Ownership 2**

In file `chatlogic.h` / `chatlogic.cpp`, adapt the vector `_nodes` in a way that the instances of `GraphNodes` to which the vector elements refer are exclusively owned by the class `ChatLogic`.

Use an appropriate type of smart pointer to achieve this. Where required, make changes to the code such that data structures and function parameters reflect the changes. When passing the `GraphNode` instances to functions, make sure to not transfer ownership and try to contain the changes to class `ChatLogic` where possible.

to achieve the goal which is posted on this task.





# Final Project

## Your Tasks

- **Task 4 : Moving Smart Pointers**

In files `chatlogic.h` / `chatlogic.cpp` and `graphnodes.h` / `graphnodes.cpp` change the ownership of all instances of `GraphEdge` in a way such that each instance of `GraphNode` exclusively owns the outgoing `GraphEdges` and holds non-owning references to incoming `GraphEdges`.

Use appropriate smart pointers and where required, make changes to the code such that data structures and function parameters reflect the changes. When transferring ownership from class `ChatLogic`, where all instances of `GraphEdge` are created, into instances of `GraphNode`, make sure to use move semantics.

So we are copying an instance of a smart pointer which lives on the stack by



# Final Project

## Your Tasks

- **Task 5 : Moving the ChatBot**

In file `chatlogic.cpp`, create a local `ChatBot` instance on the stack at the bottom of function `LoadAnswerGraphFromFile`. Then, use move semantics to pass the `ChatBot` instance into the root node. Make sure that `ChatLogic` has no ownership relation to the `ChatBot` instance and thus is no longer responsible for memory allocation and deallocation.

Note that the member `\_chatBot` remains so it can be used as a communication handle between GUI and `ChatBot` instance. Make all required changes in files `chatlogic.h` / `chatlogic.cpp` and `graphnode.h` / `graphnode.cpp`. When the program is executed, messages on which part of the Rule of Five components of `ChatBot` is called should be printed to the console.

**The last task here is to really move the ChatBot around.**



ND213.C03.FP.06 Walkthrough - Chatgui.H

```
chatgui.h x
1 #ifndef CHATGUI_H_
2 #define CHATGUI_H_
3
4 #include <wx/wx.h>
5
6 class ChatLogic; // forward declaration
7
8 // middle part of the window containing the dialog between user and chatbot
9 class ChatBotPanelDialog : public wxScrolledWindow
10 {
11 private:
12     // control elements
13     wxBoxSizer *_dialogSizer;
14     wxBitmap _image;
15
16     /** STUDENT CODE
17     */
18
19     ChatLogic *_chatLogic;
20
21     /** EOF STUDENT CODE */
22
23 public:
24     // constructor / destructor
25     ChatBotPanelDialog(wxWindow *parent, wxWindowID id);
26     ~ChatBotPanelDialog();
27
28     // getter / setter
29     ChatLogic *GetChatLogicHandle() { return _chatLogic; }
30
31     // events
32     void paintEvent(wxPaintEvent &evt);
33     void paintNow();
34     void render(wxDC &dc);
35
36     // proprietary functions
37     void AddDialogItem(wxString text, bool isFromUser = true);
38     void PrintChatbotResponse(std::string response);
39
40 };
41
42 // dialog item shown in ChatBotPanelDialog
43 class ChatBotPanelDialogItem : public wxPanel
44 {
45
46 }
```

There's an enclosing Student Code tag around each place in

ND213 C03 FP 06 Walkthrough - Chatgui.H

```
chatgui.h x
1 #ifndef CHATGUI_H_
2 #define CHATGUI_H_
3
4 #include <wx/wx.h>
5
6 class ChatLogic; // forward declaration
7
8 // middle part of the window containing the dialog between user and chatbot
9 class ChatBotPanelDialog : public wxScrolledWindow
10 {
11 private:
12     // control elements
13     wxBoxSizer *_dialogSizer;
14     wxBitmap _image;
15
16     ///// STUDENT CODE
17     /////
18     ChatLogic *_chatLogic;
19
20     /////
21     ///// EOF STUDENT CODE
22     I
23
24 public:
25     // constructor / destructor
26     ChatBotPanelDialog(wxWindow *parent, wxWindowID id);
27     ~ChatBotPanelDialog();
28
29     // getter / setter
30     ChatLogic *GetChatLogicHandle() { return _chatLogic; }
31
32     // events
33     void paintEvent(wxPaintEvent &evt);
34     void paintNow();
35     void render(wxDC &dc);
36
37     // proprietary functions
38     void AddDialogItem(wxString text, bool isFromUser = true);
39     void PrintChatbotResponse(std::string response);
40
41     DECLARE_EVENT_ID(EVT_CHATBOT_PANEL_DIALOG)
42 };
43
44 // dialog item shown in ChatbotPanelDialog
45 class ChatBotPanelDialogItem : public wxPanel
46 {
47     wxString text;
48 }
```

Student Code and End of Student Code.

ND213.C03.FP.06.Walkthrough - Chatgui.H

```
chatgui.h x
chatgui.h - MemoryManagement_FinalProjectStudentVersion (Workspace)

41     DECLARE_EVENT_TABLE()
42 }
43
44 // dialog item shown in ChatBotPanelDialog
45 class ChatBotPanelDialogItem : public wxPanel
46 {
47 private:
48     // control elements
49     wxStaticBitmap *_chatBotImg;
50     wxStaticText *_chatBotTxt;
51
52 public:
53     // constructor / destructor
54     ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser);
55 }
56
57 // frame containing all control elements
58 class ChatBotFrame : public wxFrame
59 {
59 private:
60     // control elements
61     ChatBotPanelDialog *_panelDialog;
62     wxTextCtrl *_userTextCtrl;
63
64     // events
65     void OnEnter(wxCommandEvent &WXUNUSED(event));
66
67 public:
68     ChatBotFrame(const wxString &title);
69 }
70
71 // control panel for background image display
72 class ChatBotFrameImagePanel : public wxPanel
73 {
74     // control elements
75     wxBitmap _image;
76
77 public:
78
79     // events
80     void paintEvent(wxPaintEvent &event);
81     void paintNow();
82     void render(wxDC &dc);
83 }
```

Also, we have overloaded an event here and wx widgets event.

ND213.C03.FP.06.Walkthrough - Chatgui.H

```
chatgui.h
41     DECLARE_EVENT_TABLE()
42 }
43 // dialog item shown in ChatBotPanelDialog
44 class ChatBotPanelDialogItem : public wxPanel
45 {
46     private:
47         // control elements
48         wxStaticBitmap * _chatBotImg;
49         wxStaticText * _chatBotTxt;
50
51     public:
52         // constructor / destructor
53         ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser);
54     };
55
56 // frame containing all control elements
57 class ChatBotFrame : public wxFrame
58 {
59     private:
60         // control elements
61         ChatBotPanelDialog *_panelDialog;
62         wxTextCtrl *_userTextCtrl;
63
64     public:
65         // events
66         void OnEnter(wxCommandEvent &WXUNUSED(event));
67
68     public:    I
69         // constructor / desctructor
70         ChatBotFrame(const wxString &title);
71     };
72
73 // control panel for background image display
74 class ChatBotFrameImagePanel : public wxPanel
75 {
76     private:
77         // control elements
78         wxBitmap _image;
79
80     public:
81
82         // events
83         void paintEvent(wxPaintEvent &event);
84         void paintNow();
85         void render(wxDC &dc);
86
87 }
```

OnEnter means as soon as you finish typing your query to the chatbot and press enter,

ND213 C03 FP 06 Walkthrough - Chatgui.H

```
chatgui.h x
41     DECLARE_EVENT_TABLE()
42 }
43
44 // dialog item shown in ChatBotPanelDialog
45 class ChatBotPanelDialogItem : public wxPanel
46 {
47 private:
48     // control elements
49     wxStaticBitmap *_chatBotImg;
50     wxStaticText *_chatBotTxt;
51
52 public:
53     // constructor / destructor
54     ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser);
55 }
56
57 // frame containing all control elements
58 class ChatBotFrame : public wxFrame
59 {
59
60 private:
61     // control elements
62     ChatBotPanelDialog *_panelDialog;
63     wxTextCtrl *_userTextCtrl;
64
65     // events
66     void OnEnter(wxCommandEvent &wxUNUSED(event));
67
68 public: I
69     // constructor / desctructor
70     ChatBotFrame(const wxString &title);
71 }
72
73 // control panel for background image display
74 class ChatBotFrameImagePanel : public wxPanel
75 {
76     // control elements
77     wxBitmap _image;
78
79 public:
80
81     // events
82
83     void paintEvent(wxPaintEvent &event);
84     void paintNow();
85     void render(wxDC &dc);
86
87
```

this press to the enter key is captured by the OnEvent.

ND213.C03.FP.06 Walkthrough - Chatgui.H

```
chatgui.h x
71  };
72
73 // control panel for background image display
74 class ChatBotFrameImagePanel : public wxPanel
75 {
76     // control elements
77     wxBitmap _image;
78
79 public:
80     // constructor / desctructor
81     ChatBotFrameImagePanel(wxFrame *parent);
82
83     // events
84     void paintEvent(wxPaintEvent &evt);
85     void paintNow();
86     void render(wxDC &dc);
87
88     DECLARE_EVENT_TABLE()
89 };
90
91 // wxWidgets app that hides main()
92 class ChatBotApp : public wxApp
93 {
94 public:
95     // events
96     virtual bool OnInit();
97 };
98
99 #endif /* CHATGUI_H_ */
100
```

we have the OnInit event where we start everything else.

ND213.C03.FP.06 Walkthrough - Chatgui.H

```
chatgui.h x
chatgui.h -- MemoryManagement_FinalProjectStudentVersion (Workspace)

Final Project Student ... 71  };
Final Project Student ... 72
Final Project Student ... 73 // control panel for background image display
Final Project Student ... 74 class ChatBotFrameImagePanel : public wxPanel
Final Project Student ... 75 {
Final Project Student ... 76     // control elements
Final Project Student ... 77     wxBitmap _image;
Final Project Student ... 78
Final Project Student ... 79 public:
Final Project Student ... 80     // constructor / destructor
Final Project Student ... 81     ChatBotFrameImagePanel(wxFrame *parent);
Final Project Student ... 82
Final Project Student ... 83     // events
Final Project Student ... 84     void paintEvent(wxPaintEvent &evt);
Final Project Student ... 85     void paintNow();
Final Project Student ... 86     void render(wxDC &dc);
Final Project Student ... 87
Final Project Student ... 88     DECLARE_EVENT_TABLE()
Final Project Student ... 89 };
Final Project Student ... 90
Final Project Student ... 91 // wxWidgets app that hides main()
Final Project Student ... 92 class ChatBotApp : public wxApp
Final Project Student ... 93 {
Final Project Student ... 94 public:
Final Project Student ... 95     // events
Final Project Student ... 96     virtual bool OnInit();
Final Project Student ... 97 };
Final Project Student ... 98
Final Project Student ... 99 #endif /* CHATGUI_H_ */
Final Project Student ... 100
```

encapsulates the infinite loop which runs

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspaces)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

EXPLORER

chatgui.cpp

Final Project Student... .vscode bin src answergraph.txt chatbot.cpp chatbot.h chatbot.png chatgui.cpp 2 chatgui.h chatlogic.cpp chatlogic.h graphedge.cpp graphedge.h graphnode.cpp graphnode.h sf\_bridge\_inner.jpg sf\_bridge.jpg user.png {} MemoryManagement\_Fi...

```
33     int iIndexAlloc = 1;
34     _userTextCtrl = new wxTextCtrl(ctrlPanel, idTextXtrl, "", wxDefaultPosition, wxSize(width, 50), wxTE_PROCESS_ENTER, wxDefaultValidator, wxTextCtrlNameStr);
35     Connect(idTextXtrl, wxEVT_TEXT_ENTER, wxCommandEvent::OnEnter);
36
37     // create vertical sizer for panel alignment and add panels
38     wxBoxSizer *vertBoxSizer = new wxBoxSizer(wxVERTICAL);
39     vertBoxSizer->AddSpacer(90);
40     vertBoxSizer->Add(_panelDialog, 6, wxEXPAND | wxALL, 0);
41     vertBoxSizer->Add(_userTextCtrl, 1, wxEXPAND | wxALL, 5);
42     ctrlPanel->SetSizer(vertBoxSizer);
43
44     // position window in screen center
45     this->Centre();
46 }
47
48 void ChatBotFrame::OnEnter(wxCommandEvent &WXUNUSED(event))
49 {
50     // retrieve text from text control
51     wxString userText = _userTextCtrl->GetLineText(0);
52
53     // add new user text to dialog
54     _panelDialog->AddDialogItem(userText, true);
55
56     // delete text in text control
57     _userTextCtrl->Clear();
58
59     // send user text to chatbot
60     _panelDialog->GetChatLogicHandle()->SendMessageToChatbot(std::string(userText.mb_str()));
61 }
62
63 BEGIN_EVENT_TABLE(ChatBotFrameImagePanel, wxPanel)
64 EVT_PAINT(ChatBotFrameImagePanel::paintEvent) // catch paint events
65 END_EVENT_TABLE()
66
67 ChatBotFrameImagePanel::ChatBotFrameImagePanel(wxFrame *parent) : wxPanel(parent)
68 {
69 }
70
71 void ChatBotFrameImagePanel::paintEvent(wxPaintEvent &evt)
72 {
73     render();
74 }
75
76 void ChatBotFrameImagePanel::paintNow()
77 {
78     wxClientDC dc(this);
79 }
```

and then we send the user text to the chatbot via this line here.

0:39 / 2:59

CC HD YouTube

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

EXPLORER

chatgui.cpp

Final Project Student ...

.vscode

bin

src

answergraph.txt

chatbot.cpp

chatbot.h

chatbot.png

chatgui.cpp

chatgui.h

chatlogic.cpp

chatlogic.h

graphedge.cpp

graphedge.h

graphnode.cpp

graphnode.h

sf\_bridge\_inner.jpg

sf\_bridge.jpg

user.png

{} MemoryManagement\_Fi...

98 }  
99 BEGIN\_EVENT\_TABLE(ChatBotPanelDialog, wxPanel)  
100 EVT\_PAINT(ChatBotPanelDialog::paintEvent) // catch paint events  
101 END\_EVENT\_TABLE()  
102  
103 ChatBotPanelDialog::ChatBotPanelDialog(wxWindow \*parent, wxWindowID id)  
104 : wxScrolledWindow(parent, id)  
105 {  
106 // sizer will take care of determining the needed scroll size  
107 \_dialogSizer = new wxBoxSizer(wxVERTICAL);  
108 this->SetSizer(\_dialogSizer);  
109  
110 // allow for PNG images to be handled  
111 wxInitAllImageHandlers();  
112  
113 ///// STUDENT CODE  
114 /////  
115 // create chat logic instance  
116 \_chatLogic = new ChatLogic();  
117  
118 // pass pointer to chatbot dialog so answers can be displayed in GUI  
119 \_chatLogic->SetPanelDialogHandle(this);  
120  
121 // load answer graph from file  
122 \_chatLogic->LoadAnswerGraphFromFile("answergraph.txt");  
123  
124 /////  
125 ///// EOF STUDENT CODE  
126 /////  
127 }  
128  
129 ChatBotPanelDialog::~ChatBotPanelDialog()  
130 {  
131 ///// STUDENT CODE  
132 /////  
133  
134 delete \_chatLogic;  
135  
136 }  
137  
138 }  
139  
140 void ChatBotPanelDialog::AddDialogItem(wxString text, bool isFromUser)  
141 {  
142 // add a single dialog element to the sizer  
143 }

In this case, we are instantiating the ChatLogic instance on the heap.

1:26 / 2:59

CC HD YouTube

chatgui.cpp - MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213.C03.FP.07 Walkthrough - Chatgui.Cpp

Watch later Share

EXPLORER

chatgui.cpp

Final Project Student ...

.vscode

bin

src

answergraph.txt

chatbot.cpp

chatbot.h

chatbot.png

chatgui.cpp

chatgui.h

chatlogic.cpp

chatlogic.h

graphedge.cpp

graphedge.h

graphnode.cpp

graphnode.h

sf\_bridge\_inner.jpg

sf\_bridge.jpg

user.png

{} MemoryManagement\_Fi...

98 }  
99 BEGIN\_EVENT\_TABLE(ChatBotPanelDialog, wxPanel)  
100 EVT\_PAINT(ChatBotPanelDialog::paintEvent) // catch paint events  
101 END\_EVENT\_TABLE()  
102  
103 ChatBotPanelDialog::ChatBotPanelDialog(wxWindow \*parent, wxWindowID id)  
104 : wxScrolledWindow(parent, id)  
105 {  
106 // sizer will take care of determining the needed scroll size  
107 \_dialogSizer = new wxBoxSizer(wxVERTICAL);  
108 this->SetSizer(\_dialogSizer);  
109  
110 // allow for PNG images to be handled  
111 wxInitAllImageHandlers();  
112  
113 ///// STUDENT CODE  
114 /////  
115 // create chat logic instance  
116 \_chatLogic = new ChatLogic();  
117  
118 // pass pointer to chatbot dialog so answers can be displayed in GUI  
119 \_chatLogic->SetPanelDialogHandle(this);  
120  
121 // load answer graph from file  
122 \_chatLogic->LoadAnswerGraphFromFile("answergraph.txt");  
123  
124 /////  
125 ///// EOF STUDENT CODE  
126 /////  
127 }  
128  
129 ChatBotPanelDialog::~ChatBotPanelDialog()  
130 {  
131 ///// STUDENT CODE  
132 /////  
133  
134 delete \_chatLogic;  
135  
136  
137  
138  
139 }  
140  
141 void ChatBotPanelDialog::AddDialogItem(wxString text, bool isFromUser)  
142 {  
143 // add a single dialog element to the sizer

We us then setting the handle to the panel dialogue,

1:27 / 2:59

CC HD YouTube

chatgui.cpp - MemoryManagement\_FinalProjectStudentVersion (Workspace)

ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

.vscode  
bin  
src  
answergraph.txt  
chatbot.cpp  
chatbot.h  
chatbot.png  
chatgui.cpp 2  
chatgui.h  
chatlogic.cpp  
chatlogic.h  
graphedge.cpp  
graphedge.h  
graphnode.cpp  
graphnode.h  
sf\_bridge\_inner.jpg  
sf\_bridge.jpg  
user.png  
MemoryManagement\_Fi...

```
100 BEGIN_EVENT_TABLE(ChatBotPanelDialog, wxPanel)
101 EVT_PAINT(ChatBotPanelDialog::paintEvent) // catch paint events
102 END_EVENT_TABLE()
103
104 ChatBotPanelDialog::ChatBotPanelDialog(wxWindow *parent, wxWindowID id)
105 : wxScrolledWindow(parent, id)
106 {
107     // sizer will take care of determining the needed scroll size
108     _dialogSizer = new wxBoxSizer(wxVERTICAL);
109     this->SetSizer(_dialogSizer);
110
111     // allow for PNG images to be handled
112     wxInitAllImageHandlers();
113
114     ///// STUDENT CODE
115     /////
116
117     // create chat logic instance
118     _chatLogic = new ChatLogic();
119
120     // pass pointer to chatbot dialog ChatBotPanelDialog *this is in GUI
121     _chatLogic->SetPanelDialogHandle(this);
122
123     // load answer graph from file
124     _chatLogic->LoadAnswerGraphFromFile("answergraph.txt");
125
126     /////
127     ///// EOF STUDENT CODE
128 }
129
130 ChatBotPanelDialog::~ChatBotPanelDialog()
131 {
132     ///// STUDENT CODE
133     /////
134
135     delete _chatLogic;
136
137 }
138
139 void ChatBotPanelDialog::AddDialogItem(wxString text, bool isFromUser)
140 {
141     // add a single dialog element to the sizer
142 }
```

this is the pointer to the current instance of the panel dialog,

OUTLINE 1:31 / 2:59 CC HD YouTube

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

```
EXPLORER FILE PROJECTS COMMANDS F1 PROJECTS chatgui.cpp
```

Final Project Student... .vscode bin src answergraph.txt chatbot.cpp chatbot.h chatbot.png chatgui.cpp 2 chatgui.h chatlogic.cpp chatlogic.h graphedge.cpp graphedge.h graphnode.cpp graphnode.h sf\_bridge\_inner.jpg sf\_bridge.jpg user.png

{ MemoryManagement\_Fi...

```
171 }
172 void ChatBotPanelDialog::paintNow()
173 {
174     wxClientDC dc(this);
175     render(dc);
176 }
177 }

178 void ChatBotPanelDialog::render(wxDC &dc)
179 {
180     wxImage image;
181     image.LoadFile("sf_bridge_inner.jpg");
182
183     wxSize sz = this->GetSize();
184     wxImage imgSmall = image.Rescale(sz.GetWidth(), sz.GetHeight(), wxIMAGE_QUALITY_HIGH);
185
186     _image = wxBitmap(imgSmall);
187     dc.DrawBitmap(_image ChatBotPanelDialogItem::ChatBotPanelDialogItem(<error-type> *parent, wxString text, bool isFromUser)
188
189     +2 overloads
190
191 ChatBotPanelDialogItem::ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser)
192 : wxPanel(parent, -1, wxPoint(-1, -1), wxDefaultSize, wxBORDER_NONE)
193 {
194     // retrieve image from chatbot
195     wxBitmap *bitmap = isFromUser == true ? nullptr : ((ChatBotPanelDialog*)parent)->GetChatLogicHandle()->GetImageFromChatbot();
196
197     // create image and text
198     _chatBotImg = new wxStaticBitmap(this, wxID_ANY, (isFromUser ? wxBitmap("user.png", wxBITMAP_TYPE_PNG) : *bitmap), wxDefaultPosition, wxDefaultSize);
199     _chatBotTxt = new wxStaticText(this, wxID_ANY, text, wxDefaultPosition, wxDefaultSize, wxALIGN_CENTRE | wxBORDER_NONE);
200     _chatBotTxt->SetForegroundColour(isFromUser == true ? wxColor(*wxBLACK) : wxColor(*wxWHITE));
201
202     // create sizer and add elements
203     wxBoxSizer *horzBoxSizer = new wxBoxSizer(wxHORIZONTAL);
204     horzBoxSizer->Add(_chatBotTxt, 8, wxEXPAND | wxALL, 1);
205     horzBoxSizer->Add(_chatBotImg, 2, wxEXPAND | wxALL, 1);
206     this->SetSizer(horzBoxSizer);
207
208     // wrap text after 150 pixels
209     _chatBotTxt->Wrap(150);
210
211     this->SetBackgroundColour((isFromUser == true ? wxAUXILIARY : wxWHITE));
212 }
```

this is one answer from the chatbot or one query from the user.

213 }

2:04 / 2:59

CC HD YouTube

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

```
chatgui.cpp 171 }  
172  
173 void ChatBotPanelDialog::paintNow()  
174 {  
175     wxClientDC dc(this);  
176     render(dc);  
177 }  
178  
179 void ChatBotPanelDialog::render(wxDC &dc)  
180 {  
181     wxImage image;  
182     image.LoadFile("sf_bridge_inner.jpg");  
183  
184     wxSize sz = this->GetSize();  
185     wxImage imgSmall = image.Rescale(sz.GetWidth(), sz.GetHeight(), wxIMAGE_QUALITY_HIGH);  
186  
187     _image = wxBitmap(imgSmall);  
188     dc.DrawBitmap(_image, 0, 0, false);  
189 }  
190  
191 ChatBotPanelDialogItem::ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser)  
192 : wxPanel(parent, -1, wxPoint(-1, -1), wxDefaultSize, wxBORDER_NONE)  
193 {  
194     // retrieve image from chatbot  
195     wxBitmap *bitmap = isFromUser == true ? nullptr : ((ChatBotPanelDialog*)parent)->GetChatLogicHandle()->GetImageFromChatbot();  
196  
197     // create image and text  
198     _chatBotImg = new wxStaticBitmap(this, wxID_ANY, (isFromUser ? wxBitmap("user.png", wxBITMAP_TYPE_PNG) : *bitmap), wxDefaultPosition, wxDefaultSize);  
199     _chatBotTxt = new wxStaticText(this, wxID_ANY, text, wxDefaultPosition, wxDefaultSize, wxALIGN_CENTRE | wxBORDER_NONE);  
200     _chatBotTxt->SetForegroundColour(isFromUser == true ? wxColor(*wxBLACK) : wxColor(*wxWHITE));  
201  
202     // create sizer and add elements  
203     wxBoxSizer *horzBoxSizer = new wxBoxSizer(wxHORIZONTAL);  
204     horzBoxSizer->Add(_chatBotTxt, 8, wxEXPAND | wxALL, 1);  
205     horzBoxSizer->Add(_chatBotImg, 2, wxEXPAND | wxALL, 1);  
206     this->SetSizer(horzBoxSizer);  
207  
208     // wrap text after 150 pixels  
209     _chatBotTxt->Wrap(150);  
210  
211     // Set background colour  
212     _chatBotTxt->SetBackgroundColour(isFromUser == true ? wxTInt(255, 255, 255) : wxTInt(100, 100, 100));  
213 }
```

This query is about when the message is from the user use a proprietary bitmap,

2:27 / 2:59

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

```
chatgui.cpp 171 }  
172  
173 void ChatBotPanelDialog::paintNow()  
174 {  
175     wxClientDC dc(this);  
176     render(dc);  
177 }  
178  
179 void ChatBotPanelDialog::render(wxDC &dc)  
180 {  
181     wxImage image;  
182     image.LoadFile("sf_bridge_inner.jpg");  
183  
184     wxSize sz = this->GetSize();  
185     wxImage imgSmall = image.Rescale(sz.GetWidth(), sz.GetHeight(), wxIMAGE_QUALITY_HIGH);  
186  
187     _image = wxBitmap(imgSmall);  
188     dc.DrawBitmap(_image, 0, 0, false);  
189 }  
190  
191 ChatBotPanelDialogItem::ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser)  
192 : wxPanel(parent, -1, wxPoint(-1, -1), wxDefaultSize, wxBORDER_NONE)  
193 {  
194     // retrieve image from chatbot  
195     wxBitmap *bitmap = isFromUser == true ? nullptr : ((ChatBotPanelDialog*)parent)->GetChatLogicHandle()->GetImageFromChatbot();  
196  
197     // create image and text  
198     _chatBotImg = new wxStaticBitmap(this, wxID_ANY, (isFromUser ? wxBitmap("user.png", wxBITMAP_TYPE_PNG) : *bitmap), wxDefaultPosition, wxDefaultSize);  
199     _chatBotTxt = new wxStaticText(this, wxID_ANY, text, wxDefaultPosition, wxDefaultSize, wxALIGN_CENTRE | wxBORDER_NONE);  
200     _chatBotTxt->SetForegroundColour(isFromUser == true ? wxColor(*wxBLACK) : wxColor(*wxWHITE));  
201  
202     // create sizer and add elements  
203     wxBoxSizer *horzBoxSizer = new wxBoxSizer(wxHORIZONTAL);  
204     horzBoxSizer->Add(_chatBotTxt, 8, wxEXPAND | wxALL, 1);  
205     horzBoxSizer->Add(_chatBotImg, 2, wxEXPAND | wxALL, 1);  
206     this->SetSizer(horzBoxSizer);  
207  
208     // wrap text after 150 pixels  
209     _chatBotTxt->Wrap(150);  
210  
211     // set background color  
212     this->SetBackgroundColour((isFromUser ? wxColor(*wxBLACK) : wxColor(*wxWHITE)));  
213 }
```

if it's not from the user,

Watch later Share

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

```
chatgui.cpp 171 }  
172  
173 void ChatBotPanelDialog::paintNow()  
174 {  
175     wxClientDC dc(this);  
176     render(dc);  
177 }  
178  
179 void ChatBotPanelDialog::render(wxDC &dc)  
180 {  
181     wxImage image;  
182     image.LoadFile("sf_bridge_inner.jpg");  
183  
184     wxSize sz = this->GetSize();  
185     wxImage imgSmall = image.Rescale(sz.GetWidth(), sz.GetHeight(), wxIMAGE_QUALITY_HIGH);  
186  
187     _image = wxBitmap(imgSmall);  
188     dc.DrawBitmap(_image, 0, 0, false);  
189 }  
190  
191 ChatBotPanelDialogItem::ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser)  
192 : wxPanel(parent, -1, wxPoint(-1, -1), wxDefaultSize, wxBORDER_NONE)  
193 {  
194     // retrieve image from chatbot  
195     wxBitmap *bitmap = isFromUser == true ? nullptr : ((ChatBotPanelDialog*)parent)->GetChatLogicHandle()->GetImageFromChatbot();  
196  
197     // create image and text  
198     _chatBotImg = new wxStaticBitmap(this, wxID_ANY, (isFromUser ? wxBitmap("user.png", wxBITMAP_TYPE_PNG) : *bitmap), wxDefaultPosition, wxDefaultSize);  
199     _chatBotTxt = new wxStaticText(this, wxID_ANY, text, wxDefaultPosition, wxDefaultSize, wxALIGN_CENTRE | wxBORDER_NONE);  
200     _chatBotTxt->SetForegroundColour(isFromUser == true ? wxColor(*wxBLACK) : wxColor(*wxWHITE));  
201  
202     // create sizer and add elements  
203     wxBoxSizer *horzBoxSizer = new wxBoxSizer(wxHORIZONTAL);  
204     horzBoxSizer->Add(_chatBotTxt, 8, wxEXPAND | wxALL, 1);  
205     horzBoxSizer->Add(_chatBotImg, 2, wxEXPAND | wxALL, 1);  
206     this->SetSizer(horzBoxSizer);  
207  
208     // wrap text after 150 pixels  
209     _chatBotTxt->Wrap(150);  
210  
211     // SetBackgroundColour(isFromUser == true ? wxTE_YELLOW : wxTE_DARK);  
212 }  
213 }
```

if it's from the chatbot use the data behind this handle,

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

```
chatgui.cpp 171 }  
172  
173 void ChatBotPanelDialog::paintNow()  
174 {  
175     wxClientDC dc(this);  
176     render(dc);  
177 }  
178  
179 void ChatBotPanelDialog::render(wxDC &dc)  
180 {  
181     wxImage image;  
182     image.LoadFile("sf_bridge_inner.jpg");  
183  
184     wxSize sz = this->GetSize();  
185     wxImage imgSmall = image.Rescale(sz.GetWidth(), sz.GetHeight(), wxIMAGE_QUALITY_HIGH);  
186  
187     _image = wxBitmap(imgSmall);  
188     dc.DrawBitmap(_image, 0, 0, false);  
189 }  
190  
191 ChatBotPanelDialogItem::ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser)  
192 : wxPanel(parent, -1, wxPoint(-1, -1), wxDefaultSize, wxBORDER_NONE)  
193 {  
194     // retrieve image from chatbot  
195     wxBitmap *bitmap = isFromUser == true ? nullptr : ((ChatBotPanelDialog*)parent)->GetChatLogicHandle()->GetImageFromChatbot();  
196  
197     // create image and text  
198     _chatBotImg = new wxStaticBitmap(this, wxID_ANY, (isFromUser ? wxBitmap("user.png", wxBITMAP_TYPE_PNG) : *bitmap), wxPoint(-1, -1), wxDefaultSize);  
199     _chatBotTxt = new wxStaticText(this, wxID_ANY, text, wxPoint(-1, -1), wxDefaultSize, wxALIGN_CENTRE | wxBORDER_NONE);  
200     _chatBotTxt->SetForegroundColour(isFromUser == true ? wxColor(*wxBLACK) : wxColor(*wxWHITE));  
201  
202     // create sizer and add elements  
203     wxBoxSizer *horzBoxSizer = new wxBoxSizer(wxHORIZONTAL);  
204     horzBoxSizer->Add(_chatBotTxt, 8, wxEXPAND | wxALL, 1);  
205     horzBoxSizer->Add(_chatBotImg, 2, wxEXPAND | wxALL, 1);  
206     this->SetSizer(horzBoxSizer);  
207  
208     // wrap text after 150 pixels  
209     _chatBotTxt->Wrap(150);  
210  
211     // set background  
212     this->SetBackgroundColour(isFromUser == true ? wxColor(*wxLIGHTCYAN) : wxColor(*wxLIGHTMAGENTA));  
213 }
```

which directly leads through the image,

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

```
chatgui.cpp 171 }  
172  
173 void ChatBotPanelDialog::paintNow()  
174 {  
175     wxClientDC dc(this);  
176     render(dc);  
177 }  
178  
179 void ChatBotPanelDialog::render(wxDC &dc)  
180 {  
181     wxImage image;  
182     image.LoadFile("sf_bridge_inner.jpg");  
183  
184     wxSize sz = this->GetSize();  
185     wxImage imgSmall = image.Rescale(sz.GetWidth(), sz.GetHeight(), wxIMAGE_QUALITY_HIGH);  
186  
187     _image = wxBitmap(imgSmall);  
188     dc.DrawBitmap(_image, 0, 0, false);  
189 }  
190  
191 ChatBotPanelDialogItem::ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser)  
192 : wxPanel(parent, -1, wxPoint(-1, -1), wxDefaultSize, wxBORDER_NONE)  
193 {  
194     // retrieve image from chatbot  
195     wxBitmap *bitmap = isFromUser == true ? nullptr : ((ChatBotPanelDialog*)parent)->GetChatLogicHandle()->GetImageFromChatbot();  
196  
197     // create image and text  
198     _chatBotImg = new wxStaticBitmap(this, wxID_ANY, (isFromUser ? wxBitmap("user.png", wxBITMAP_TYPE_PNG) : *bitmap), wxDefaultPosition, wxDefaultSize);  
199     _chatBotTxt = new wxStaticText(this, wxID_ANY, text, wxDefaultPosition, wxDefaultSize, wxALIGN_CENTRE | wxBORDER_NONE);  
200     _chatBotTxt->SetForegroundColour(isFromUser == true ? wxColor(*wxBLACK) : wxColor(*wxWHITE));  
201  
202     // create sizer and add elements  
203     wxBoxSizer *horzBoxSizer = new wxBoxSizer(wxHORIZONTAL);  
204     horzBoxSizer->Add(_chatBotTxt, 8, wxEXPAND | wxALL, 1);  
205     horzBoxSizer->Add(_chatBotImg, 2, wxEXPAND | wxALL, 1);  
206     this->SetSizer(horzBoxSizer);  
207  
208     // wrap text after 150 pixels  
209     _chatBotTxt->Wrap(150);  
210  
211     // set background colour  
212     this->SetBackgroundColour(isFromUser == true ? wxColor(*wxLIGHTCYAN) : wxColor(*wxDARKBLUE));  
213 }
```

which is managed by the moving chatbot,

OUTLINE

chatgui.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 07 Walkthrough - Chatgui.Cpp

Watch later Share

```
chatgui.cpp 171 }  
172  
173 void ChatBotPanelDialog::paintNow()  
174 {  
175     wxClientDC dc(this);  
176     render(dc);  
177 }  
178  
179 void ChatBotPanelDialog::render(wxDC &dc)  
180 {  
181     wxImage image;  
182     image.LoadFile("sf_bridge_inner.jpg");  
183  
184     wxSize sz = this->GetSize();  
185     wxImage imgSmall = image.Rescale(sz.GetWidth(), sz.GetHeight(), wxIMAGE_QUALITY_HIGH);  
186  
187     _image = wxBitmap(imgSmall);  
188     dc.DrawBitmap(_image, 0, 0, false);  
189 }  
190  
191 ChatBotPanelDialogItem::ChatBotPanelDialogItem(wxPanel *parent, wxString text, bool isFromUser)  
192 : wxPanel(parent, -1, wxPoint(-1, -1), wxDefaultSize, wxBORDER_NONE)  
193 {  
194     // retrieve image from chatbot  
195     wxBitmap *bitmap = isFromUser == true ? nullptr : ((ChatBotPanelDialog*)parent)->GetChatLogicHandle()->GetImageFromChatbot();  
196  
197     // create image and text  
198     _chatBotImg = new wxStaticBitmap(this, wxID_ANY, (isFromUser ? wxBitmap("user.png", wxBITMAP_TYPE_PNG) : *bitmap), wxDefaultPosition, wxDefaultSize);  
199     _chatBotTxt = new wxStaticText(this, wxID_ANY, text, wxDefaultPosition, wxDefaultSize, wxALIGN_CENTRE | wxBORDER_NONE);  
200     _chatBotTxt->SetForegroundColour(isFromUser == true ? wxColor(*wxBLACK) : wxColor(*wxWHITE));  
201  
202     // create sizer and add elements  
203     wxBoxSizer *horzBoxSizer = new wxBoxSizer(wxHORIZONTAL);  
204     horzBoxSizer->Add(_chatBotTxt, 8, wxEXPAND | wxALL, 1);  
205     horzBoxSizer->Add(_chatBotImg, 2, wxEXPAND | wxALL, 1);  
206     this->SetSizer(horzBoxSizer);  
207  
208     // wrap text after 150 pixels  
209     _chatBotTxt->Wrap(150);  
210  
211     // set background colour  
212     this->SetBackgroundColour(isFromUser == true ? wxPINK : wxLIMEGREEN);  
213 }
```

which is somewhere in the graph network.

ND213.C03.FP.08 Walkthrough - Chatlogic.H

```
chatlogic.h
1 // chatlogic.h
2
3 #include "chatbot.h"
4
5 class ChatBot;
6
7 class GraphEdge;
8
9 class GraphNode;
10
11 class ChatLogic
12 {
13     // forward declarations
14
15     private:
16         // STUDENT CODE
17
18         // data handles (owned)
19         std::vector<GraphNode*> _nodes;
20         std::vector<GraphEdge*> _edges;
21
22         // STUDENT CODE
23
24         // EOF STUDENT CODE
25
26         // data handles (not owned)
27         GraphNode *_currentNode;
28         ChatBot *_chatBot;
29         ChatBotPanelDialog *_panelDialog;
30
31         // proprietary type definitions
32         typedef std::vector<std::pair<std::string, std::string>> tokenlist;
33
34         // proprietary functions
35         template <typename T>
36         void AddAllTokensToElement(std::string tokenID, tokenlist &tokens, T &element);
37
38     public:
39         // constructor / destructor
40         ChatLogic();
41         ~ChatLogic();
42
43         // getter / setter
44         void SetPanelDialogHandle(ChatBotPanelDialog *panelDialog);
45             SetChatbotHandle(ChatBot *chatbot);
46
47
48         void LoadAnswerGraphFromFile(std::string filename);
49         void SendMessageToChatbot(std::string message);
50         void SendMessageToUser(std::string message);
51         wxBitmap *GetImageFromChatbot();
52 }
```

Now the ownership of edges lies with the chatlogic and you will

ND213.C03.FP.08 Walkthrough - Chatlogic.H

```
chatlogic.h
1 // chatlogic.h
2
3 #include "chatbot.h"
4
5 class ChatBot;
6
7 class GraphEdge;
8
9 class GraphNode;
10
11 class ChatLogic
12 {
13     private:
14         // STUDENT CODE
15         // data handles (owned)
16         std::vector<GraphNode *> _nodes;
17         std::vector<GraphEdge *> _edges;
18
19         /////
20         // EOF STUDENT CODE
21
22         // data handles (not owned)
23         GraphNode *_currentNode;
24         ChatBot *_chatBot;
25         ChatBotPanelDialog *_panelDialog;
26
27         // proprietary type definitions
28         typedef std::vector<std::pair<std::string, std::string>> tokenlist;
29
30         // proprietary functions
31         template <typename T>
32         void AddAllTokensToElement(std::string tokenID, tokenlist &tokens, T &element);
33
34     public:
35         // constructor / destructor
36         ChatLogic();
37         ~ChatLogic();
38
39         // getter / setter
40         void SetPanelDialogHandle(ChatBotPanelDialog *panelDialog);
41
42         // Search for a node by ID
43         GraphNode *SearchGraphNodeByID(std::string chatbot);
44
45         // Load answer graph from file
46         void LoadAnswerGraphFromFile(std::string filename);
47
48         // Send message to chatbot
49         void SendMessageToChatbot(std::string message);
50
51         // Send message to user
52         void SendMessageToUser(std::string message);
53         wxBitmap *GetImageFromChatbot();
54 }
```

change this and one of the tasks who will come to this pointers.

ND213.C03.FP.08 Walkthrough - Chatlogic.H

```
#include "chatgui.h"
// forward declarations
class ChatBot;
class GraphEdge;
class GraphNode;

class ChatLogic
{
private:
    // STUDENT CODE
    std::vector<GraphNode *> _nodes;
    // data handles (owned) purposefully not versioned
    std::vector<GraphNode *> _nodes;
    std::vector<GraphEdge *> _edges;
    // EOF STUDENT CODE

    // data handles (not owned)
    GraphNode *_currentNode;
    ChatBot *_chatBot;
    ChatBotPanelDialog *_panelDialog;

    // proprietary type definitions
    typedef std::vector<std::pair<std::string, std::string>> tokenlist;

    // proprietary functions
    template <typename T>
    void AddAllTokensToElement(std::string tokenID, tokenlist &tokens, T &element);

public:
    // constructor / destructor
    ChatLogic();
    ~ChatLogic();

    // getter / setter
    void SetPanelDialogHandle(ChatBotPanelDialog *panelDialog);
    Setchatbotandialoghandle(chatbot, paneldialog);

    void LoadAnswerGraphFromFile(std::string filename);
    void SendMessageToChatbot(std::string message);
    Void SendMessageToUser(std::string message);
    wxBitmap *GetImageFromChatbot();
};

0:36 / 1:21
```

So where the ownership of edges is moved into some of the nodes here,

ND213.C03.FP.08 Walkthrough - Chatlogic.H

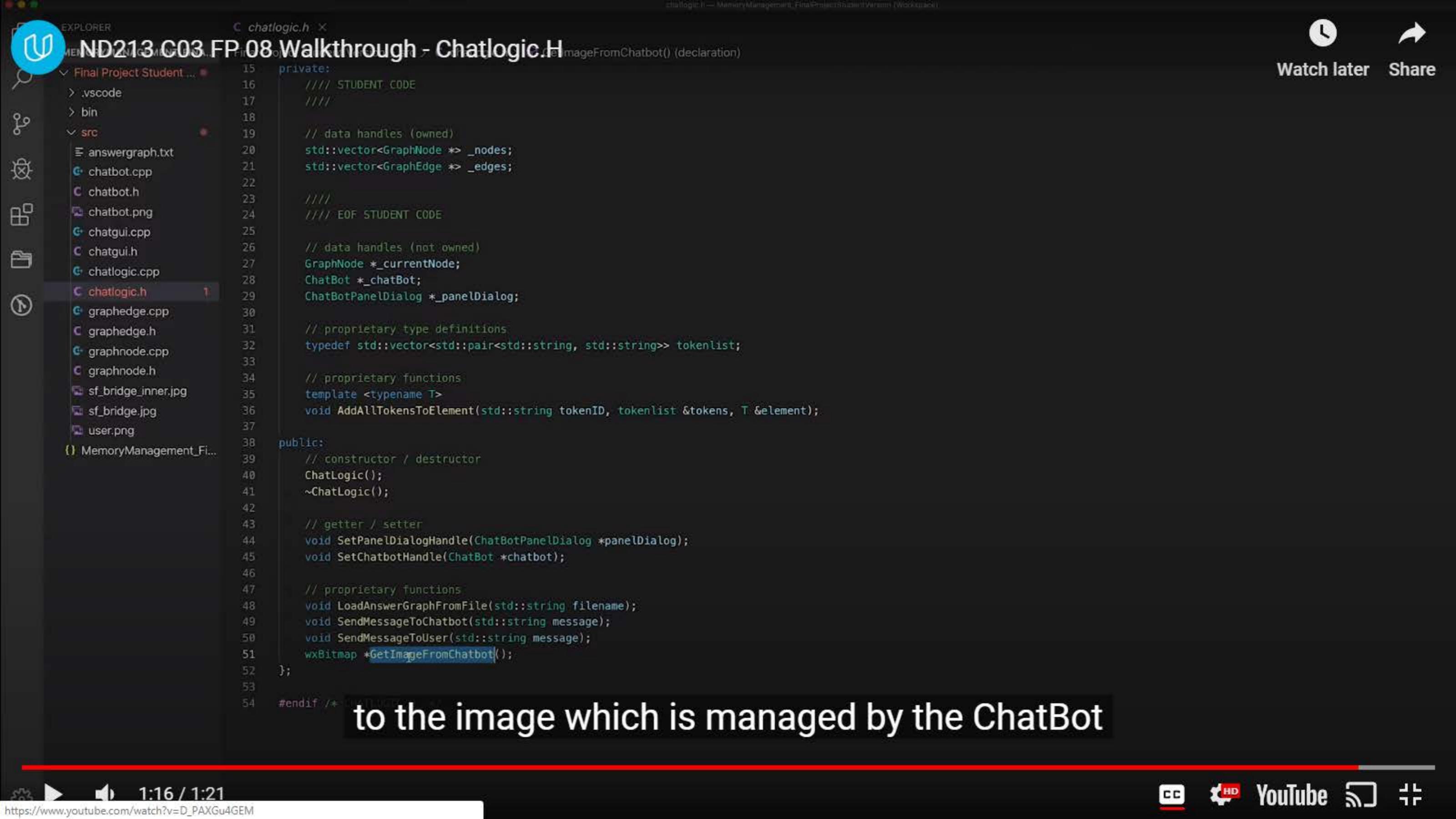
```
chatlogic.h 15 private:
16     // STUDENT CODE
17
18
19     // data handles (owned)
20     std::vector<GraphNode *> _nodes;
21     std::vector<GraphEdge *> _edges;
22
23
24     // EOF STUDENT CODE
25
26     // data handles (not owned)
27     GraphNode *_currentNode;
28     ChatBot *_chatBot;
29     ChatBotPanelDialog *_panelDialog;
30
31     // proprietary type definitions
32     typedef std::vector<std::pair<std::string, std::string>> tokenlist;
33
34     // proprietary functions
35     template <typename T>
36     void AddAllTokensToElement(std::string tokenID, tokenlist &tokens, T &element);
37
38 public:
39     // constructor / destructor
40     ChatLogic();
41     ~ChatLogic();
42
43     // getter / setter
44     void SetPanelDialogHandle(ChatBotPanelDialog *panelDialog);
45     void SetChatbotHandle(ChatBot *chatbot);
46
47     // proprietary functions
48     void LoadAnswerGraphFromFile(std::string filename);
49     void SendMessageToChatbot(std::string message);
50     void SendMessageToUser(std::string message);
51     wxBitmap *GetImageFromChatbot();
52 }
53
```

from the ChatBot and we have this function here which retrieves the handle

ND213 C03 FP 08 Walkthrough - Chatlogic.H

```
chatlogic.h x
chatlogic.h 15 private:
chatlogic.h 16     /// STUDENT CODE
chatlogic.h 17     ///
chatlogic.h 18
chatlogic.h 19     // data handles (owned)
chatlogic.h 20     std::vector<GraphNode *> _nodes;
chatlogic.h 21     std::vector<GraphEdge *> _edges;
chatlogic.h 22
chatlogic.h 23     ///
chatlogic.h 24     /// EOF STUDENT CODE
chatlogic.h 25
chatlogic.h 26     // data handles (not owned)
chatlogic.h 27     GraphNode *_currentNode;
chatlogic.h 28     ChatBot *_chatBot;
chatlogic.h 29     ChatBotPanelDialog *_panelDialog;
chatlogic.h 30
chatlogic.h 31     // proprietary type definitions
chatlogic.h 32     typedef std::vector<std::pair<std::string, std::string>> tokenlist;
chatlogic.h 33
chatlogic.h 34     // proprietary functions
chatlogic.h 35     template <typename T>
chatlogic.h 36     void AddAllTokensToElement(std::string tokenID, tokenlist &tokens, T &element);
chatlogic.h 37
chatlogic.h 38 public:
chatlogic.h 39     // constructor / destructor
chatlogic.h 40     ChatLogic();
chatlogic.h 41     ~ChatLogic();
chatlogic.h 42
chatlogic.h 43     // getter / setter
chatlogic.h 44     void SetPanelDialogHandle(ChatBotPanelDialog *panelDialog);
chatlogic.h 45     void SetChatbotHandle(ChatBot *chatbot);
chatlogic.h 46
chatlogic.h 47     // proprietary functions
chatlogic.h 48     void LoadAnswerGraphFromFile(std::string filename);
chatlogic.h 49     void SendMessageToChatbot(std::string message);
chatlogic.h 50     void SendMessageToUser(std::string message);
chatlogic.h 51     wxBitmap *GetImageFromChatbot();
chatlogic.h 52 }
chatlogic.h 53
chatlogic.h 54 #endif /* !defined(CHATLOGIC_H_ */
```

to the image which is managed by the ChatBot



ND213.C03.FP.08 Walkthrough - Chatlogic.H

```
chatlogic.h:15 private: // STUDENT CODE
chatlogic.h:16     //-
chatlogic.h:17
chatlogic.h:18
chatlogic.h:19     // data handles (owned)
chatlogic.h:20     std::vector<GraphNode *> _nodes;
chatlogic.h:21     std::vector<GraphEdge *> _edges;
chatlogic.h:22
chatlogic.h:23
chatlogic.h:24     //-
chatlogic.h:25     // EOF STUDENT CODE
chatlogic.h:26
chatlogic.h:27     // data handles (not owned)
chatlogic.h:28     GraphNode *_currentNode;
chatlogic.h:29     ChatBot *_chatBot;
chatlogic.h:30     ChatBotPanelDialog *_panelDialog;
chatlogic.h:31
chatlogic.h:32     // proprietary type definitions
chatlogic.h:33     typedef std::vector<std::pair<std::string, std::string>> tokenlist;
chatlogic.h:34
chatlogic.h:35     // proprietary functions
chatlogic.h:36     template <typename T>
chatlogic.h:37     void AddAllTokensToElement(std::string tokenID, tokenlist &tokens, T &element);
chatlogic.h:38
chatlogic.h:39     public:
chatlogic.h:40         // constructor / destructor
chatlogic.h:41         ChatLogic();
chatlogic.h:42         ~ChatLogic();
chatlogic.h:43
chatlogic.h:44         // getter / setter
chatlogic.h:45         void SetPanelDialogHandle(ChatBotPanelDialog *panelDialog);
chatlogic.h:46         void SetChatbotHandle(ChatBot *chatbot);
chatlogic.h:47
chatlogic.h:48         // proprietary functions
chatlogic.h:49         void LoadAnswerGraphFromFile(std::string filename);
chatlogic.h:50         void SendMessageToChatbot(std::string message);
chatlogic.h:51         void SendMessageToUser(std::string message);
chatlogic.h:52         wxBitmap *GetImageFromChatbot();
chatlogic.h:53
chatlogic.h:54 #endif
```

so it can be displayed in the graphical user interface

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

# ND213.C03.FP10 Walkthrough - Graphnode.H

Watch later Share

```
// forward declarations
class GraphEdge;

class GraphNode
{
private:
    // STUDENT CODE
    // data handles (owned)
    std::vector<GraphEdge *> _childEdges; // edges to subsequent nodes

    // data handles (not owned) I
    std::vector<GraphEdge *> _parentEdges; // edges to preceding nodes
    ChatBot *_chatBot;

    // EOF STUDENT CODE

    // proprietary members
    int _id;
    std::vector<std::string> _answers;

public:
    // constructor / destructor
    GraphNode(int id);
    ~GraphNode();

    // getter / setter
    int GetID() { return _id; }
    int GetNumberOfChildEdges() { return _childEdges.size(); }
    GraphEdge *GetChildEdgeAtIndex(int index);
    std::vector<std::string> GetAnswers() { return _answers; }
    int GetNumberOfParents() { return _parentEdges.size(); }

    // proprietary functions
    void AddToken(std::string token); // add answers to list
    void AddEdgeToParentNode(GraphEdge *edge);
    void AddEdgeToChildNode(GraphEdge *edge);

    // STUDENT CODE
    // ...
};

void MoveChatbotHere(ChatBot *chatbot);
// ...
```

the Edges will or some Edges,

graphnode.h — MemoryManagement/FinalProjectStudentVersion/Walkthrough

# ND213 C03 FP10 Walkthrough - Graphnode.H

Watch later Share

```
// forward declarations
class GraphEdge;

class GraphNode
{
private:
    // STUDENT CODE
    std::vector<GraphEdge *> GraphNode::_childEdges;
    // data handles (owned) purposefully not versioned
    std::vector<GraphEdge *> _childEdges; // edges to subsequent nodes

    // data handles (not owned)
    std::vector<GraphEdge *> _parentEdges; // edges to preceding nodes
    ChatBot *_chatBot;

    // EOF STUDENT CODE

    // proprietary members
    int _id;
    std::vector<std::string> _answers;

public:
    // constructor / destructor
    GraphNode(int id);
    ~GraphNode();

    // getter / setter
    int GetID() { return _id; }
    int GetNumberOfChildEdges() { return _childEdges.size(); }
    GraphEdge *GetChildEdgeAtIndex(int index);
    std::vector<std::string> GetAnswers() { return _answers; }
    int GetNumberOfParents() { return _parentEdges.size(); }

    // proprietary functions
    void AddToken(std::string token); // add answers to list
    void AddEdgeToParentNode(GraphEdge *edge);
    void AddEdgeToChildNode(GraphEdge *edge);

    // STUDENT CODE
    /////

    void MoveChatbotHere(ChatBot *chatbot);
    /////
}
```

the childEdges at least,

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

# ND213-C03.FP10.Walkthrough - Graphnode.H

Watch later Share

```
// forward declarations
class GraphEdge;

class GraphNode
{
private:
    // STUDENT CODE
    //

    // data handles (owned)
    std::vector<GraphEdge *> _childEdges; // edges to subsequent nodes

    // data handles (not owned)
    std::vector<GraphEdge *> _parentEdges; // edges to preceding nodes
    ChatBot *_chatBot;

    //
    // EOF STUDENT CODE

    // proprietary members
    int _id;
    std::vector<std::string> _answers;

public:
    // constructor / destructor
    GraphNode(int id);
    ~GraphNode();

    // getter / setter
    int GetID() { return _id; }
    int GetNumberOfChildEdges() { return _childEdges.size(); }
    GraphEdge *GetChildEdgeAtIndex(int index);
    std::vector<std::string> GetAnswers() { return _answers; }
    int GetNumberOfParents() { return _parentEdges.size(); }

    // proprietary functions
    void AddToken(std::string token); // add answers to list
    void AddEdgeToParentNode(GraphEdge *edge);
    void AddEdgeToChildNode(GraphEdge *edge);

    // STUDENT CODE
    //

void MoveChatbotHere(ChatBot *chatbot);
    //

}
```

they will be owned by a graphnode.

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP10 Walkthrough - Graphnode.H

```
// forward declarations
class GraphEdge;
class GraphNode
{
private:
    // STUDENT CODE
    // data handles (owned)
    std::vector<GraphEdge*> _childEdges; // edges to subsequent nodes
    ChatBot *_chatBot;
    std::vector<GraphEdge*> _parentEdges; // edges to preceding nodes
    ChatBot *_chatbot;

    // EOF STUDENT CODE
    // proprietary members
    int _id;
    std::vector<std::string> _answers;

public:
    // constructor / destructor
    GraphNode(int id);
    ~GraphNode();

    // getter / setter
    int GetID() { return _id; }
    int GetNumberOfChildEdges() { return _childEdges.size(); }
    GraphEdge *GetChildEdgeAtIndex(int index);
    std::vector<std::string> GetAnswers() { return _answers; }
    int GetNumberOfParents() { return _parentEdges.size(); }

    // proprietary functions
    void AddToken(std::string token); // add answers to list
    void AddEdgeToParentNode(GraphEdge *edge);
    void AddEdgeToChildNode(GraphEdge *edge);

    // STUDENT CODE
    void MoveChatbotHere(ChatBot *chatbot);
    //
```

This is going to be null or null pointer if



Watch later Share

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

# ND213 C03 FP10 Walkthrough - Graphnode.H

graphnode.h

```
// forward declarations
class GraphEdge;
class GraphNode
{
private:
    // STUDENT CODE
    // data handles (owned)
    std::vector<GraphEdge*> _childEdges; // edges to subsequent nodes
    ChatBot *_chatBot
    // data h
    std::vector<GraphEdge*> _parentEdges; // edges to preceding nodes
    ChatBot *_chatBot;

    // EOF STUDENT CODE
    // proprietary members
    int _id;
    std::vector<std::string> _answers;

public:
    // constructor / destructor
    GraphNode(int id);
    ~GraphNode();

    // getter / setter
    int GetID() { return _id; }
    int GetNumberOfChildEdges() { return _childEdges.size(); }
    GraphEdge *GetChildEdgeAtIndex(int index);
    std::vector<std::string> GetAnswers() { return _answers; }
    int GetNumberOfParents() { return _parentEdges.size(); }

    // proprietary functions
    void AddToken(std::string token); // add answers to list
    void AddEdgeToParentNode(GraphEdge *edge);
    void AddEdgeToChildNode(GraphEdge *edge);

    // 
    void MoveChatbotHere(ChatBot *chatbot);
    //
```

the chatBot is not currently residing within this node.

EXPLORER

SEARCH

RECENT DOCUMENTS

Find

src

Final Project Student...

.vscode

bin

answergraph.txt

chatbot.cpp

chatbot.h

chatbot.png

chatgui.cpp

chatgui.h

chatlogic.cpp

chatlogic.h

graphedge.cpp

graphedge.h

graphnode.cpp

graphnode.h

sf\_bridge\_inner.jpg

sf\_bridge.jpg

user.png

MemoryManagement\_Fi...

0:55 / 2:03

Watch later

Share

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

# ND213-C03.FP10 Walkthrough - Graphnode.H

graphnode.h x

EXPLORER

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

Final Project Student... .vscode bin src answergraph.txt chatbot.cpp chatbot.h chatbot.png chatgui.cpp chatgui.h chatlogic.cpp chatlogic.h graphedge.cpp graphedge.h graphnode.cpp graphnode.h sf\_bridge\_inner.jpg sf\_bridge.jpg user.png

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

```
graphnode.h
9 // forward declarations
10 class GraphEdge;
11
12 class GraphNode
13 {
14 private:
15     // STUDENT CODE
16     /////
17
18     // data handles (owned)
19     std::vector<GraphEdge *> _childEdges; // edges to subsequent nodes
20
21     ChatBot *_chatBot
22     std::vector<GraphEdge *> _parentEdges; // edges to preceding nodes
23     ChatBot *_chatBot;
24
25     /////
26     // EOF STUDENT CODE
27
28     // proprietary members
29     int _id;
30     std::vector<std::string> _answers;
31
32 public:
33     // constructor / destructor
34     GraphNode(int id);
35     ~GraphNode();
36
37     // getter / setter
38     int GetID() { return _id; }
39     int GetNumberOfChildEdges() { return _childEdges.size(); }
40     GraphEdge *GetChildEdgeAtIndex(int index);
41     std::vector<std::string> GetAnswers() { return _answers; }
42     int GetNumberOfParents() { return _parentEdges.size(); }
43
44     // proprietary functions
45     void AddToken(std::string token); // add answers to list
46     void AddEdgeToParentNode(GraphEdge *edge);
47     void AddEdgeToChildNode(GraphEdge *edge);
48
49     // STUDENT CODE
50     /////
51
52     void MoveChatbotHere(ChatBot *chatbot);
53
54     /////
55
```

Watch later Share

As soon as it is assigned to this node,

0:58 / 2:03

CC HD YouTube

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Workfile)

EXPLORER

ND213-C03.FP10.Walkthrough - Graphnode.H

Watch later Share

graphnode.h

```
graphnode.h x
C graphnode.h x
graphnode.h — MemoryManagement_FinalProjectStudentVersion (Workfile)

graphnode.h
Final Project Student ... 9 // forward declarations
src 10 class GraphEdge;
answergraph.txt 11
chatbot.cpp 12 class GraphNode
chatbot.h 13 {
chatbot.png 14     private:
chatgui.cpp 15         /// STUDENT CODE
chatgui.h 16         ///
graphedge.cpp 17             // data handles (owned)
graphedge.h 18                 std::vector<GraphEdge*> _childEdges; // edges to subsequent nodes
chatlogic.cpp 19                     ChatBot *_chatBot
chatlogic.h 20                         // data h
sf_bridge_inner.jpg 21                 std::vector<GraphEdge*> _parentEdges; // edges to preceding nodes
sf_bridge.jpg 22                     ChatBot *_chatBots;
user.png 23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
```

// proprietary members

int \_id;

std::vector<std::string> \_answers;

public:

// constructor / destructor

GraphNode(int id);

~GraphNode();

// getter / setter

int GetID() { return \_id; }

int GetNumberOfChildEdges() { return \_childEdges.size(); }

GraphEdge \*GetChildEdgeAtIndex(int index);

std::vector<std::string> GetAnswers() { return \_answers; }

int GetNumberOfParents() { return \_parentEdges.size(); }

// proprietary functions

void AddToken(std::string token); // add answers to list

void AddEdgeToParentNode(GraphEdge \*edge);

AddEdgeToChildNode(GraphEdge \*edge);

void MoveChatbotHere(ChatBot \*chatbot);

///

this is going to be a valid memory address and when

1:00 / 2:03

CC HD YouTube

graphnode.h — MemoryManagement\_FinalProjectStudentVersion (Workfile)

Udacity 3.C03.FP.10 Walkthrough - Graphnode.H

Watch later Share

graphnode.h

```
graphnode.h x
C graphnode.h x
graphnode.h — MemoryManagement_FinalProjectStudentVersion (Workfile)

9 // forward declarations
10 class GraphEdge;
11
12 class GraphNode
13 {
14 private:
15     // STUDENT CODE
16
17     // data handles (owned)
18     std::vector<GraphEdge *> _childEdges; // edges to subsequent nodes
19
20     ChatBot *_chatBot
21
22     std::vector<GraphEdge *> _parentEdges; // edges to preceding nodes
23     ChatBot *_chatBot;
24
25
26     // EOF STUDENT CODE
27
28     // proprietary members
29     int _id;
30     std::vector<std::string> _answers;
31
32 public:
33     // constructor / destructor
34     GraphNode(int id);
35     ~GraphNode();
36
37     // getter / setter
38     int GetID() { return _id; }
39     int GetNumberOfChildEdges() { return _childEdges.size(); }
40     GraphEdge *GetChildEdgeAtIndex(int index);
41     std::vector<std::string> GetAnswers() { return _answers; }
42     int GetNumberOfParents() { return _parentEdges.size(); }
43
44     // proprietary functions
45     void AddToken(std::string token); // add answers to list
46     void AddEdgeToParentNode(GraphEdge *edge);
47     void AddEdgeToChildNode(GraphEdge *edge);
48
49     // STUDENT CODE
50
51
52     void MoveChatbotHere(ChatBot *chatbot);
53
54     //
```

chatBot moves away, it's invalidated again.

1:02 / 2:03

CC HD YouTube

graphnode.h

```
graphnode.h x
ND213-C03.FP10.Walkthrough - Graphnode.H
graphnode.h — MemoryManagement_FinalProjectStudentVersion (Workspaces)

EXPLORER
MENUS & COMMANDS F10: Go to file
graphnode.h
Final Project Student ... 25 /////
Final Project Student ... 26 //EOF STUDENT CODE
Final Project Student ... 27
Final Project Student ... 28 // proprietary members
Final Project Student ... 29 int _id;
Final Project Student ... 30 std::vector<std::string> _answers;
Final Project Student ... 31
Final Project Student ... 32 public:
Final Project Student ... 33 // constructor / destructor
Final Project Student ... 34 GraphNode(int id);
Final Project Student ... 35 ~GraphNode();
Final Project Student ... 36
Final Project Student ... 37 // getter / setter
Final Project Student ... 38 int GetID() { return _id; }
Final Project Student ... 39 int GetNumberOfChildEdges() { return _childEdges.size(); }
Final Project Student ... 40 GraphEdge *GetChildEdgeAtIndex(int index);
Final Project Student ... 41 std::vector<std::string> GetAnswers() { return _answers; }
Final Project Student ... 42 int GetNumberOfParents() { return _parentEdges.size(); }
Final Project Student ... 43
Final Project Student ... 44 // proprietary functions
Final Project Student ... 45 void AddToken(std::string token); // add answers to list
Final Project Student ... 46 void AddEdgeToParentNode(GraphEdge *edge);
Final Project Student ... 47 void AddEdgeToChildNode(GraphEdge *edge);
Final Project Student ... 48 /////
Final Project Student ... 49 // STUDENT CODE
Final Project Student ... 50 /////
Final Project Student ... 51
Final Project Student ... 52 void MoveChatbotHere(ChatBot *chatbot);
Final Project Student ... 53
Final Project Student ... 54 /////
Final Project Student ... 55 //EOF STUDENT CODE
Final Project Student ... 56
Final Project Student ... 57 void MoveChatbotToNewNode(GraphNode *newNode);
Final Project Student ... 58 };
Final Project Student ... 59
Final Project Student ... 60 #endif /* GRAPHNODE_H_ */
```

A token would be an answer.

graphnode.cpp — MemoryManagement\_FinalProject\_StudentVersion (Workspace)

# ND213 C03 FP 11 Walkthrough - Graphnode.Cpp

Watch later Share

EXPLORER

graphnode.cpp

Final Project Student ...

.vscode

bin

src

answergraph.txt

chatbot.cpp

chatbot.h

chatbot.png

chatgui.cpp

chatgui.h

chatlogic.cpp

chatlogic.h

graphedge.cpp

graphedge.h

graphnode.cpp

graphnode.h

sf\_bridge\_inner.jpg

sf\_bridge.jpg

user.png

MemoryManagement\_Fi...

```
20 void GraphNode::AddToken(std::string token)
21 {
22     _answers.push_back(token);
23 }
24
25 void GraphNode::AddEdgeToParentNode(GraphEdge *edge)
26 {
27     _parentEdges.push_back(edge);
28 }
29
30 void GraphNode::AddEdgeToChildNode(GraphEdge *edge)
31 {
32     _childEdges.push_back(edge);
33 }
34
35 // STUDENT CODE
36 //
37 void GraphNode::MoveChatbotHere(ChatBot *chatbot)
38 {
39     _chatBot = chatbot;
40     _chatBot->SetCurrentNode(this);
41 }
42
43 void GraphNode::MoveChatbotToNewNode(GraphNode *newNode)
44 {
45     newNode->MoveChatbotHere(_chatBot);
46     _chatBot = nullptr; // invalidate pointer at source
47 }
48 //
49 // EOF STUDENT CODE
50
51 GraphEdge *GraphNode::GetChildEdgeAtIndex(int index)
52 {
53     // STUDENT CODE
54     //
55
56     return _childEdges[index];
57 }
58 //
59 // EOF STUDENT CODE
60 }
```

Once we execute this line here,

1:42 / 2:15

CC HD YouTube

graphnode.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 11 Walkthrough - Graphnode.Cpp

EXPLORER

graphnode.cpp

Final Project Student ...

- .vscode
- bin
- src
  - answergraph.txt
  - chatbot.cpp
  - chatbot.h
  - chatbot.png
  - chatgui.cpp
  - chatgui.h
  - chatlogic.cpp
  - chatlogic.h
  - graphedge.cpp
  - graphedge.h
  - graphnode.cpp
  - graphnode.h
  - sf\_bridge\_inner.jpg
  - sf\_bridge.jpg
  - user.png
- MemoryManagement\_Fi...

graphnode.cpp

```
20 void GraphNode::AddToken(std::string token)
21 {
22     _answers.push_back(token);
23 }
24
25 void GraphNode::AddEdgeToParentNode(GraphEdge *edge)
26 {
27     _parentEdges.push_back(edge);
28 }
29
30 void GraphNode::AddEdgeToChildNode(GraphEdge *edge)
31 {
32     _childEdges.push_back(edge);
33 }
34
35 // STUDENT CODE
36 /**
37 void GraphNode::MoveChatbotHere(ChatBot *chatbot)
38 {
39     _chatBot = chatbot;
40     _chatBot->SetCurrentNode(this);
41 }
42
43 void GraphNode::MoveChatbotToNewNode(GraphNode *newNode)
44 {
45     newNode->MoveChatbotHere(_chatBot);
46     _chatBot = nullptr; // invalidate pointer at source
47 }
48
49 // EOF STUDENT CODE
50
51 GraphEdge *GraphNode::GetChildEdgeAtIndex(int index)
52 {
53     // STUDENT CODE
54 /**
55
56     return _childEdges[index];
57
58 /**
59 // EOF STUDENT CODE
60 }
```

we invalidate the chat bot pointer

Watch later Share

1:44 / 2:15

CC HD YouTube

graphnode.cpp — MemoryManagement\_FinalProjectStudentVersion (Workspace)

# ND213 C03 FP 11 Walkthrough - Graphnode.Cpp

Watch later Share

EXPLORER

graphnode.cpp

Final Project Student ...

- .vscode
- bin
- src
  - answergraph.txt
  - chatbot.cpp
  - chatbot.h
  - chatbot.png
  - chatgui.cpp
  - chatgui.h
  - chatlogic.cpp
  - chatlogic.h
  - graphedge.cpp
  - graphedge.h
  - graphnode.cpp
  - graphnode.h
  - sf\_bridge\_inner.jpg
  - sf\_bridge.jpg
  - user.png
- MemoryManagement\_Fi...

```
20 void GraphNode::AddToken(std::string token)
21 {
22     _answers.push_back(token);
23 }
24
25 void GraphNode::AddEdgeToParentNode(GraphEdge *edge)
26 {
27     _parentEdges.push_back(edge);
28 }
29
30 void GraphNode::AddEdgeToChildNode(GraphEdge *edge)
31 {
32     _childEdges.push_back(edge);
33 }
34
35 // STUDENT CODE
36 /**
37 void GraphNode::MoveChatbotHere(ChatBot *chatbot)
38 {
39     _chatBot = chatbot;
40     _chatBot->SetCurrentNode(this);
41 }
42
43 void GraphNode::MoveChatbotToNewNode(GraphNode *newNode)
44 {
45     newNode->MoveChatbotHere(_chatBot);
46     _chatBot = nullptr; // invalidate pointer at source
47 }
48
49 // EOF STUDENT CODE
50
51 GraphEdge *GraphNode::GetChildEdgeAtIndex(int index)
52 {
53     // STUDENT CODE
54 /**
55
56     return _childEdges[index];
57 }
58 */

which is at the node which we are currently leaving with a chat bot.
```

1:47 / 2:15

CC HD YouTube

graphedge.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

# ND213 C03 FP 12 Walkthrough - Graphedge

Watch later Share

```
graphedge.h ×
```

EXPLORER

Final Project Student ...

- > .vscode
- > bin
- < src
  - answergraph.txt
  - chatbot.cpp
  - chatbot.h
  - chatbot.png
  - chatgui.cpp
  - chatgui.h
  - chatlogic.cpp
  - chatlogic.h
  - graphedge.cpp
  - graphedge.h 1
  - graphnode.cpp
  - graphnode.h
  - sf\_bridge\_inner.jpg
  - sf\_bridge.jpg
  - user.png
- (.) MemoryManagement\_Fi...

```
1 #ifndef GRAPHEDGE_H_
2 #define GRAPHEDGE_H_
3
4 #include <vector>
5 #include <string>
6
7 class GraphNode; // forward declaration
8
9 class GraphEdge
10 {
11     private:
12         // data handles (not owned)
13         GraphNode *_childNode;
14         GraphNode *_parentNode;
15
16         // proprietary members
17         int _id;
18         std::vector<std::string> _keywords; // list of topics associated with this edge
19
20     public:
21         // constructor / destructor
22         GraphEdge(int id);
23
24         // getter / setter
25         int GetID() { return _id; }
26         void SetChildNode(GraphNode *childNode);
27         void SetParentNode(GraphNode *parentNode);
28         GraphNode *GetChildNode() { return _childNode; }
29         std::vector<std::string> GetKeywords() { return _keywords; }
30
31         // proprietary functions
32         void AddToken(std::string token);
33     };
34
35
36 #endif /* GRAPHEDGE_H_ */
```

So edges don't take on ownership of anything.

0:12 / 0:58

CC HD YouTube

graphedge.h — MemoryManagement\_FinalProjectStudentVersion (Walkthrough)

# ND213 C03 FP 12 Walkthrough - Graphedge

Watch later Share

EXPLORER

graphedge.h

Final Project Student ...

.vscode

bin

src

answergraph.txt

chatbot.cpp

chatbot.h

chatbot.png

chatgui.cpp

chatgui.h

chatlogic.cpp

chatlogic.h

graphedge.cpp

graphedge.h

graphnode.cpp

graphnode.h

sf\_bridge\_inner.jpg

sf\_bridge.jpg

user.png

MemoryManagement\_Fi...

```
1 #ifndef GRAPHEDGE_H_
2 #define GRAPHEDGE_H_
3
4 #include <vector>
5 #include <string>
6
7 class GraphNode; // forward declaration
8
9 class GraphEdge
10 {
11     private:
12         // data handles (not owned)
13         GraphNode *_childNode;
14         GraphNode *_parentNode;
15
16     // proprietary members
17     int _id;
18     std::vector<std::string> _keywords; // list of topics associated with this edge
19
20 public:
21     // constructor / destructor
22     GraphEdge(int id);
23
24     // getter / setter
25     int GetID() { return _id; }
26     void SetChildNode(GraphNode *childNode);
27     void SetParentNode(GraphNode *parentNode);
28     GraphNode *GetChildNode() { return _childNode; }
29     std::vector<std::string> GetKeywords() { return _keywords; }
30
31     // proprietary functions
32     void AddToken(std::string token);
33 };
34
35
36#endif /* GRAPHEDGE_H_ */
```

They are simply intermediaries between two nodes, child node,

0:17 / 0:58

CC HD YouTube

graphedge.h x

ND213 C03 FP 12 Walkthrough - Graphedge

Watch later Share

```
1 #ifndef GRAPHEDGE_H_
2 #define GRAPHEDGE_H_
3
4 #include <vector>
5 #include <string>
6
7 class GraphNode; // forward declaration
8
9 class GraphEdge
10 {
11     private:
12         // data handles (not owned)
13         GraphNode * _parentNode
14         GraphNode * _parentNode;
15
16         // proprietary members
17         int _id;
18         std::vector<std::string> _keywords; // list of topics associated with this edge
19
20     public:
21         // constructor / destructor
22         GraphEdge(int id);
23
24         // getter / setter
25         int GetID() { return _id; }
26         void SetChildNode(GraphNode *childNode);
27         void SetParentNode(GraphNode *parentNode);
28         GraphNode *GetChildNode() { return _childNode; }
29         std::vector<std::string> GetKeywords() { return _keywords; }
30
31         // proprietary functions
32         void AddToken(std::string token);
33     };
34
35
36 #endif /* GRAPHEDGE_H_ */
```

and parent node, and we have some proprietary members,

0:18 / 0:58

CC HD YouTube

ND213 C03 FP14 Walkthrough - Chatbot.Cpp

```
chatbot.cpp — MemoryManagement_FinalProjectStudentVersion (Workspace)
```

DEBUG ▶ BU ⚙ □ chatbot.cpp x

Local Static Global Registers CALL STACK PAUSED ON EXCEPTION ChatBot::~ChatBot() ... ChatBot::~ChatBot() ... GraphNode::~GraphNode() GraphNode::~GraphNode() ChatLogic::~ChatLogic() ChatLogic::~ChatLogic() ChatBotPanelDialog::~Cho ChatBotPanelDialog::~Cho ChatBotPanelDialog::~Cho wxWindowBase::Destroy() wxWindowBase::DestroyChi wxWindow::~wxWindow() wxNavigationEnabled<wxWi wxPanelBase::~wxPanelBas wxPanel::~wxPanel() ... ChatBotFrameImagePanel:: ChatBotFrameImagePanel::

Exception has occurred.  
EXC\_BAD\_ACCESS (code=EXC\_I386\_GPFLT)

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL

Launching: /Users/ahaja/Dropbox/Nebentätigkeit/Seminare/Udacity/Courses/C++ ND/Memory Management Course/Code/Final Project Student Version/bin/chatbot  
Stop reason: EXC\_BAD\_ACCESS (code=EXC\_I386\_GPFLT)

an exception has occurred.

BREAKPOINTS

C++ on throw C++ on catch 3:55 / 4:26 MODULES

Watch later Share

The screenshot shows a C++ IDE interface with a code editor, debugger, and terminal. The code editor displays a file named `chatbot.cpp` with the following content:

```
ND213 C03 FP14 Walkthrough - Chatbot.Cpp
chatbot.cpp — MemoryManagement_FinalProjectStudentVersion (Workspace)

32
33     ChatBot::~ChatBot()
34     {
35         std::cout << "ChatBot Destructor" << std::endl;
36
37         // deallocate heap memory
38         if(_image != NULL) // Attention: wxWidgets used NULL and not nullptr
39         {
40             delete _image;
41             _image = NULL;
42         }
43     }
44
45     // STUDENT CODE
46
47
48     // EOF STUDENT CODE
49
50
```

The debugger window shows a call stack with the following entries:

- 1... PAUSED ON EXCEPTION
- ChatBot::~ChatBot()
- ChatBot::~ChatBot()
- GraphNode::~GraphNode()
- GraphNode::~GraphNode()
- ChatLogic::~ChatLogic()
- ChatLogic::~ChatLogic()
- ChatBotPanelDialog::~Cho
- ChatBotPanelDialog::~Cho
- ChatBotPanelDialog::~Cho
- wxWindowBase::Destroy()
- wxWindowBase::DestroyChi
- wxWindow::~wxWindow()
- wxNavigationEnabled<wxWi
- wxPanelBase::~wxPanelBas
- wxPanel::~wxPanel()
- ChatBotFrameImagePanel::
- ChatBotFrameImagePanel::

An exception has occurred at line 40:

```
Exception has occurred.  
EXC_BAD_ACCESS (code=EXC_I386_GPFLT)
```

The terminal window shows the following output:

```
Launching: /Users/ahaja/Dropbox/Nebentaeigkeit/Seminare/Udacity/Courses/C++ ND/Memory Management Course/Code/Final Project Student Version/bin/chatbot  
Stop reason: EXC_BAD_ACCESS (code=EXC_I386_GPFLT)
```

A large text overlay at the bottom of the screen reads:

the crash doesn't occur and the valid exit code zero is printed to the console.

Watch later Share

## Task Details

In file `chatgui.h` / `chatgui.cpp`, make `_chatLogic` an exclusive resource to class `ChatbotPanelDialog` using an appropriate smart pointer. Where required, make changes to the code such that data structures and function parameters reflect the new structure.

## Task Details

In file `chatbot.h` / `chatbot.cpp`, make changes to the class `ChatBot` such that it complies with the Rule of Five. Make sure to properly allocate / deallocate memory resources on the heap and also copy member data where it makes sense to you. In each of the methods (e.g. the copy constructor), print a string of the type „`ChatBot Copy Constructor`” to the console so that you can see which method is called in later examples.

## Task Details

In file `chatlogic.h` / `chatlogic.cpp`, adapt the vector `_nodes` in a way that the instances of `GraphNodes` to which the vector elements refer are exclusively owned by the class `ChatLogic`. Use an appropriate type of smart pointer to achieve this. Where required, make changes to the code such that data structures and function parameters reflect the changes. When passing the `GraphNode` instances to functions, make sure to not transfer ownership and try to contain the changes to class `ChatLogic` where possible.

## Task Details

In files `chatlogic.h` / `chatlogic.cpp` and `graphnodes.h` / `graphnodes.cpp` change the ownership of all instances of `GraphEdge` in a way such that each instance of `GraphNode` exclusively owns the outgoing `GraphEdges` and holds non-owning references to incoming `GraphEdges`. Use appropriate smart pointers and where required, make changes to the code such that data structures and function parameters reflect the changes. When transferring ownership from class `ChatLogic`, where all instances of `GraphEdge` are created, into instances of `GraphNode`, make sure to use move semantics.

In file `chatlogic.cpp`, create a local `ChatBot` instance on the stack at the bottom of function `LoadAnswerGraphFromFile`. Then, use move semantics to pass the `ChatBot` instance into the root node. Make sure that `ChatLogic` has no ownership relation to the `ChatBot` instance and thus is no longer responsible for memory allocation and deallocation. Note that the member `_chatBot` remains so it can be used as a communication handle between GUI and `ChatBot` instance. Make all required changes in files `chatlogic.h` / `chatlogic.cpp` and `graphnode.h` / `graphnode.cpp`. When the program is executed, messages on which part of the Rule of Five components of `ChatBot` is called should be printed to the console. When sending a query to the `ChatBot`, the output should look like the following:

```
ChatBot Constructor
ChatBot Move Constructor
ChatBot Move Assignment Operator
ChatBot Destructor
ChatBot Destructor
```

# Memory Management Chatbot

## Quality of Code

CRITERIA	MEETS SPECIFICATIONS
Is the code functional?	The code compiles and runs with <code>cmake</code> and <code>make</code> .

## Task 1: Exclusive Ownership 1

CRITERIA	MEETS SPECIFICATIONS
<code>_chatLogic</code> is an exclusive resource of <code>ChatbotPanelDialog</code>	In file <code>chatgui.h</code> / <code>chatgui.cpp</code> , <code>_chatLogic</code> is made an exclusive resource to class <code>ChatbotPanelDialog</code> using an appropriate smart pointer. Where required, changes are made to the code such that data structures and function parameters reflect the new structure.

## Task 2: The Rule of Five

CRITERIA	MEETS SPECIFICATIONS
Class design meets the Rule of Five guidelines.	In file <code>chatbot.h</code> / <code>chatbot.cpp</code> , changes are made to the class <code>ChatBot</code> such that it complies with the Rule of Five. Memory resources are properly allocated / deallocated on the heap and member data is copied where it makes sense. In each of the methods (e.g. the copy constructor), a string of the type "ChatBot Copy Constructor" is printed to the console so that it is possible to see which method is called in later examples.

### Task 3: Exclusive Ownership 2

CRITERIA	MEETS SPECIFICATIONS
<p>The <code>GraphNode</code>s in the vector <code>_nodes</code> are exclusively owned by the class <code>ChatLogic</code>.</p>	<p>In file <code>chatlogic.h</code> / <code>chatlogic.cpp</code>, the vector <code>_nodes</code> are adapted in a way that the instances of <code>GraphNodes</code> to which the vector elements refer are exclusively owned by the class <code>ChatLogic</code>. An appropriate type of smart pointer is used to achieve this.</p>
<p><code>GraphNode</code> ownership is not transferred when passing instances.</p>	<p>When passing the <code>GraphNode</code> instances to functions, ownership is not transferred.</p>

## Task 4: Moving Smart Pointers

### CRITERIA

### MEETS SPECIFICATIONS

`GraphNode`'s exclusively own the outgoing `GraphEdges` and hold non-owning references to incoming `GraphEdges`.

In files `chatlogic.h` / `chatlogic.cpp` and `graphnodes.h` / `graphnodes.cpp` all instances of `GraphEdge` are changed in a way such that each instance of `GraphNode` exclusively owns the outgoing `GraphEdges` and holds non-owning references to incoming `GraphEdges`. Appropriate smart pointers are used to do this. Where required, changes are made to the code such that data structures and function parameters reflect the changes.

Move semantics are used when transferring ownership from class `ChatLogic` into instances of `GraphNode`.

In files `chatlogic.h` / `chatlogic.cpp` and `graphnodes.h` / `graphnodes.cpp`, move semantics are used when transferring ownership from class `ChatLogic`, where all instances of `GraphEdge` are created, into instances of `GraphNode`.

## Task 5: Moving the ChatBot

CRITERIA	MEETS SPECIFICATIONS
Move semantics are used correctly with <code>ChatBot</code> .	In file <code>chatlogic.cpp</code> , a local <code>ChatBot</code> instance is created on the stack at the bottom of function <code>LoadAnswerGraphFromFile</code> and move semantics are used to pass the <code>ChatBot</code> instance into the root node.
<code>chatLogic</code> has no ownership relation to the <code>ChatBot</code> instance.	<code>ChatLogic</code> has no ownership relation to the <code>ChatBot</code> instance and thus is no longer responsible for memory allocation and deallocation.
The <code>Chatbot</code> prints output to indicate Rule of Five components.	When the program is executed, messages are printed to the console indicating which Rule of Five component of <code>ChatBot</code> is being called.