University of Glasgow

Systems Programming

Introduction

Anna Lito Michala, Yehia Elkhatib

Email AnnaLito.Michala@Glasgow.ac.uk

Yehia.Elkhatib@glasgow.ac.uk

University of Glasgow

Systems Programming (H)

Lecturers

- **Anna Lito Michala** http://www.dcs.gla.ac.uk/~amichala/
  - Room: 405, Sir Alwyn Williams Building
  - AnnaLito.Michala@Glasgow.ac.uk

- **Yehia Elkhatib** https://yelkhatib.github.io/
  - Room: S322a, Sir Alwyn Williams Building
  - Yehia.Elkhatib@glasgow.ac.uk

Schedule

- **Lecture:** each Monday 15:00 – 17:00 First hour offline, second hour on Zoom
- **Lab:** starting next week, Monday 09:00 – 13:00 on Zoom **& BO 720**

Learning Outcomes

1. Demonstrate competence in **low-level systems programming**
2. **Design, implement and explain the behaviour** of low-level programs written in a systems language
3. Explain the concepts of **stack and heap allocation, pointers**, and **memory ownership**, including demonstrating an understanding of issues such as **memory leaks, buffer overflows, use-after-free,** and **race conditions**
4. Explain how **data structures are represented**, and how this interacts with caching and virtual memory, and to be able to demonstrate the performance impact of such issues
5. Discuss and reason about **concurrency, race conditions**, and the **system memory model**
6. **Build** well-structured **concurrent systems programs** of moderate size, using libraries and static analysis tools appropriately.

In this course we will use C and C++ as they are the two most important systems programming languages

Course Overview and Schedule

- **Full schedule on moodle page with instructions!**
- **Schedule Highlights**

– No lab this week (27/09), we start next week.

– No lecture or lab sheet in week 7 (beginning 01/11) as CW1 due
Use the lecture & lab time to work on CW1.

– **CW1 submission on 04/11 at 16:30pm**

– Online Guest lecture on 29/11 (TBC).

– **CW2 submission on 25/11 at 16:30pm**

– Revision Lecture in Semester 2; Exam in April/May

University of Glasgow

Assessment

- There will be **two pieces of coursework** each worth **10% of your mark**
- First coursework will be released end of next week
- Second coursework will be released mid November
- The **exam in April / May is 80% of your mark**
- There is a Lab exercise each week for improving your C programming skills. You can also get assistance with your coursework during the Lab and ask questions

Remote learning – lectures

- **Lecture delivery plan: MAY BE ADAPTED AS WE LEARN!**

– 1st hour offline/asynchronous:

- Watch videos, complete tasks and/or quiz that might be required for that day before attending the zoom.
- Can be done any time before Monday 16:00

– 2nd hour online: Meet in Zoom every Monday 16:00

- Each Zoom session will cover the topics of that week's lecture

University of Glasgow

Hybrid learning – labs

– Offline. Work for 30 .. 60 minutes on the lab sheet before attending the Zoom. Can be done any time before Monday 09:00 am.

– Online (Teams) or in person Monday 09:00 am starting from next week.

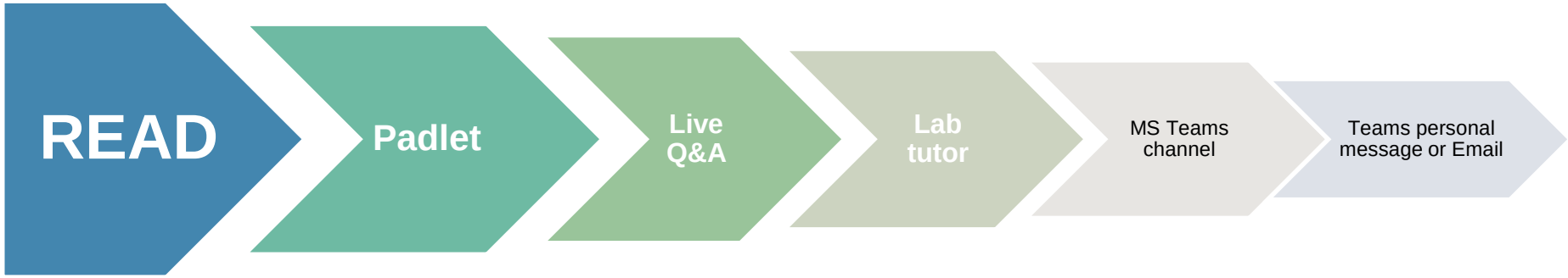
- Based on your LAB you will be allocated a specific Teams Meeting.
- Announcements will be sent by email & on Teams on which tutor will cover your group.
- Follow the link of the specific meeting to attend the online Lab.
- On alternate weeks you will be attending in person labs

Remote learning – Communication

- **Zoom sessions**

– We aim to address all questions in the Q&A session. Do not use zoom chat. Questions will be answered with **live coding** if possible

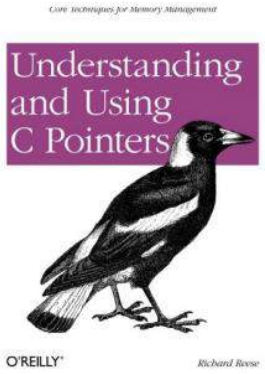
– We definitely need your help so **ASK Questions!** We will find the answer together by trying it out and writing programs



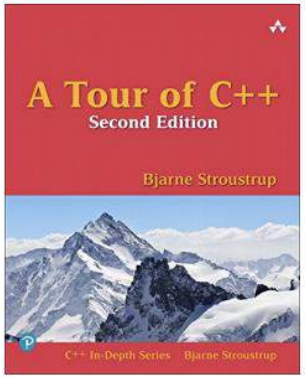
– Only email us ONLY with questions regarding personal matters

Reading Material

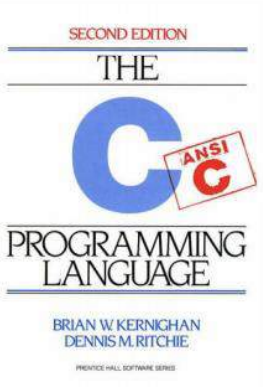
- No book required for this course (**programming** by **yourself *is the key*** to understanding)
- Two introductory recommendations (left) and the two defined treatments of C and C++ (right):



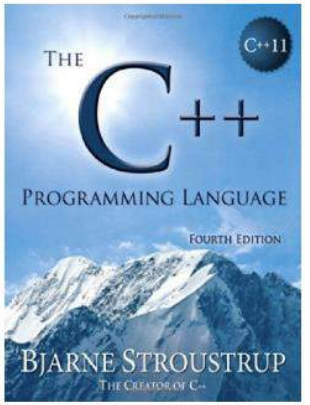
Focus on most challenging part of C: memory management



Good overview of the most important features of C++
Also available for free at
isocpp.org/tour



Reference by creators of C
Still good reference today



In-depth (1376 pages) treatment of C++ by the creator of the language

- For basic and syntactic questions use the free C Programming Wikibook and cppreference.com

University of Glasgow

What next?

- **Watch today's video lectures (Week 1: Introduction)**
- **Complete the first quiz (What languages do you speak? Experience and Expectations)**
- **Reminder: No lab today**
- **If you are already a C wizard try to work on CW1! If you can complete the AVL version this week you can skip the first 3 lectures. However, you will need to attend Lecture 4 onwards.**

University of Glasgow

Systems Programming
Introduction to C

Anna Lito Michala

Email AnnaLito.Michala@Glasgow.ac.uk

University of Glasgow

Systems Programming and System Software?

- *Systems programming* is the activity of writing computer *system software*
- In contrast we call other software *application software*
- Examples of system software:
  operating system, web browser, video games, scientific computing applications and libraries, ...
- System software has (often) particular performance constrains such as:
  fast execution time,
  low memory consumption,
  low energy usage, ...
- To achieve these performance constrains systems programming languages allow for a more fine grained **control** over the execution of programs

Teaching of fundamentals not a Language

- You will learn C and a bit of C++ as part of this course, but this is not the main goal
- I want to you to deepen your understanding of fundamental *principles* and *techniques* in *computer systems*
  - **Memory** and **Computation** as fundamental resources of computing
  - **Representation of data structures** in memory and the role of **data types**
  - **Techniques for management** of computational resources
  - Reasoning about **concurrent** systems

Understanding these principles and techniques well will make you a better computer scientist no matter what area your are interested in

University of Glasgow

History of Systems Programming Languages – 1

1950s:

Software wasn't distinguished between system and application

A single application always used the entire machine

Early programming languages were invented such as Fortran, LISP and COBOL

Grace Hopper wrote one of the first *compilers* to automatically turn the human readable program into machine code

Grace Hopper at the UNIVAC I

University of Glasgow

History of Systems Programming Languages – 2

until the 1970s: System software is written in processor specific assembly languages

1970s: *Dennis Ritchie* and *Ken Thompson* wanted to port UNIX from the PDP-7 to the PDP-11

They looked for a portable programming language and tried a language called B but then invented C as a portable *imperative* language supporting *structured* programming

PDP-7 minicomputer

Ken Thompson (sitting) and Dennis Ritchie (standing) at a PDP-11 minicomputer (Picture by Peter Hamer)

History of Systems Programming Languages – 3

1980s: *Bjarne Stroustrup* aims to enrich C with new abstraction mechanisms and creates C++

A major influence is the first *object-oriented* programming language `Simula`

Bjarne Stroustrup, the creator of C++

2010s: New systems programming languages appear. **`Rust`** (2010) and **`Swift`** (2014) are successful examples which include many *functional* programming language features

University of Glasgow

Programming Paradigms

- In *imperative* programming computations are sequences of statements that change the program's state

```
x = 41
x = x + 1
```

- *Structured* programming organises programs with subroutines and structured control flow constructs

```
sum = 0
for x in array:
    sum += x
```

- *Object-oriented* programming organises programs into objects containing data and encapsulate behaviour

```
animal = Dog()
animal.makeNoice()
```

- In *functional* programming programs are mathematical functions and avoid explicit change of state

```
fib = lambda x, x_1=1, x_2=0: x_2 if x == 0 else fib(x-1, x_1 + x_2, x_1)
```

University of Glasgow

Some questions about a simple Python program

```
x = 41
x = x + 1
```

- What value does x hold at the end of the program execution? (not a tricky question)
  - Answer: 42
- How much memory does Python take to store x? (much more tricky question)
  - Answer: In depends on the Pythton implementation.
    `sys.getsizeof(x)` gives the answer. On my machine: 28 bytes (* 8 = 224 bits)
    We can see the defining C code here
- How many instructions does Python execute to compute x + 1? (even more tricky question)
  - Answer: I don't know, but many more than just the addition ...
    Python is a dynamically typed language, so the data type of x could change at any time.

    Every operation tests the data types of the operands to check which instruction to execute.
    The C implementation for the add operation starts here

University of Glasgow

Some questions about a simple C program

```c
#include <stdio.h>
int main() {
    int x = 41;
    x = x + 1;
    printf("%d\n", x);
}
```

Online version of the code

- What value does x have at the end of the program execution? (not a tricky question)
  - Answer: 42
- How much memory does C take to store x? (not that tricky any more)
  - Answer: `sizeof(int)` usually 4 bytes (* 8 = 32 bits)
- How many instructions does C take to compute `x + 1`? (not that tricky any more)
  - Answer: 1 `add` instruction and 2 memory (`mov`) instructions

This level of control allows strong reasoning about the programs performance and execution behaviour

University of Glasgow

C by example: Fibonacci

Fibonacci recursively:

```c
int fib(int x, int x1, int x2) {
    if (x == 0) {
        return x2;
    } else {
        return fib(x - 1, x1 + x2, x1);
    }
}

int main() {
    int fib_6 = fib(6, 0, 1);
}
```

Fibonacci iteratively:

```c
int fib(int x) {
  int x1 = 0;
  int x2 = 1;

  while (x > 1) {
    int xtmp = x1 + x2;
    x1 = x2;
    x2 = xtmp;
    x = x - 1;
  }
  return x2;
}

int main() {
    int fib_6 = fib(6);
}
```

University of Glasgow

Compiling a C program

- To execute a C program we must first ***compile*** it into machine executable code
- `Java, Haskell` (and many other languages) are also compiled languages
- The ***compiler*** translates the C source code in multiple steps into machine executable code:
  1. The ***preprocessor*** expands macros
  – (e.g. `#include <stdio.h>` or `#define PI 3.14`)
  2. In the compiler stage, the source code is a) ***parsed*** and turned into an ***intermediate representation***, b) machine-specific ***assembly*** code is generated, and, finally, c) ***machine code*** is generated in an ***object file***
  3. The ***linker*** combines multiple object files into an executable
- In this course we are using the `clang` compiler.
- To compile and then execute a C program run:

```
clang source.c -o program
./program
```

University of Glasgow

1. Preprocessing

Input program as C source code

```c
#include <stdio.h>

int main() {
  int x = 41;
  x = x + 1;
  printf("%d \n", x);
}
```

Program after preprocessing

```c
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
// ...
int printf(const char * restrict, ...);
// ...

int main() {
  int x = 41;
  x = x + 1;
  printf("%d \n", x);
}
```

You can generate the code after the preprocessor stage with the `-E` flag:

```
clang source.c -E -o source.e
```

University of Glasgow

2a. Compiler intermediate representation

Program after preprocessing

```c
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
// ...
int printf(const char * restrict, ...);
// ...

int main() {
  int x = 41;
  x = x + 1;
  printf("%d \n", x);
}
```

You can generate the intermediate representation with the `-emit-llvm -S` flags:

```
clang source.c -emit-llvm -S -o source.llvm
```

Program in compiler intermediate representation

```
; ModuleID = 'source.c'
source_filename = "source.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

@.str = private unnamed_addr constant [5 x i8] c"%d \0A\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 41, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = add nsw i32 %2, 1
  store i32 %3, i32* %1, align 4
  %4 = load i32, i32* %1, align 4
  %5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str, i32
  ret i32 0
}

declare i32 @printf(i8*, ...) #1

attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-mat
attributes #1 = { "correctly-rounded-divide-sqrt-fp-math"="false" "disable-tail-calls"="false

!llvm.module.flags = !{!0, !1}
!llvm.ident = !{!2}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{!"Apple LLVM version 10.0.0 (clang-1000.11.45.2)"}
```

University of Glasgow

2b. Assembly code

Program in compiler intermediate representation

```
; ModuleID = 'source.c'
source_filename = "source.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.13.0"

@.str = private unnamed_addr constant [5 x i8] c"%d \0A\00", align 1

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 41, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  %3 = add nsw i32 %2, 1
  store i32 %3, i32* %1, align 4
  %4 = load i32, i32* %1, align 4
  %5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i
  ret i32 0
}

...
```

You can generate the assembly code with the `-S` flag:

```
clang source.c -S -o source.s
```

Program in machine-specific assembly code (here for x86-64)

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 13
    .globl    _main                     ## -- Begin function main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    L_.str(%rip), %rdi
    movl    $41, -4(%rbp)
    movl    -4(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %esi
    movb    $0, %al
    callq    _printf
    xorl    %esi, %esi
    movl    %eax, -8(%rbp)          ## 4-byte Spill
    movl    %esi, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc                    ## -- End function
    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
    .asciz    "%d \n"
.subsections_via_symbols
```

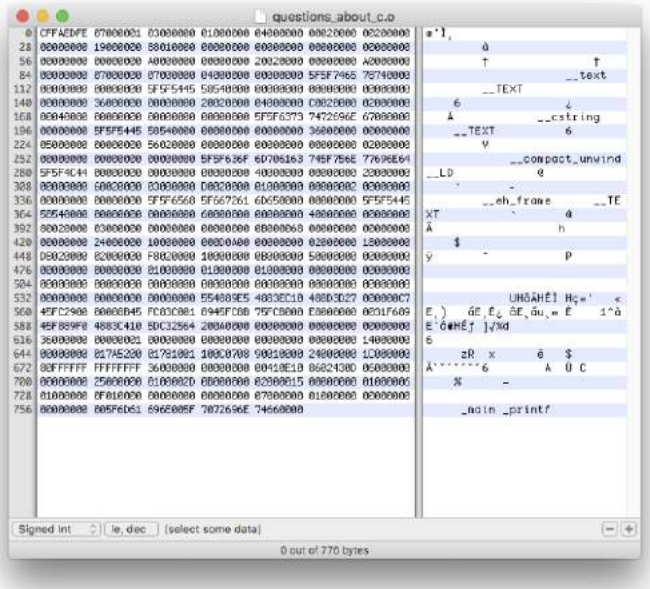
University of Glasgow

2c. Machine code

Program in machine-specific assembly code (here for x86-64)

```
    .section    __TEXT,__text,regular,pure_instructions
    .macosx_version_min 10, 13
    .globl    _main                     ## -- Begin function main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    L_.str(%rip), %rdi
    movl    $41, -4(%rbp)
    movl    -4(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %esi
    movb    $0, %al
    callq    _printf
    xorl    %esi, %esi
    movl    %eax, -8(%rbp)          ## 4-byte Spill
    movl    %esi, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc                    ## -- End function
    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
    .asciz    "%d \n"
.subsections_via_symbols
```

Program in machine (or *object*) code (here for x86-64)



You can generate an object file with machine code using the -c flag:

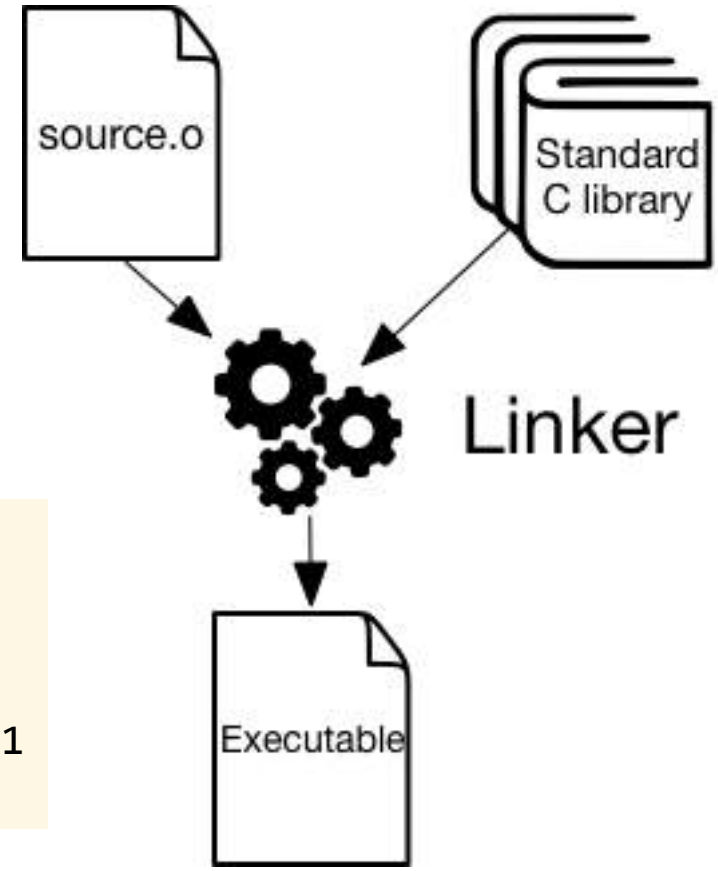
```
clang source.c -c -o source.o
```

University of Glasgow

Linking

- The linker combines one or more object files into a single executable
- The linker checks that all functions called in the program have machine code avaialble (e.g. `printf`'s machine code is in the C standard library)
- If the machine code for a function can not be found the linker complains:

```
Undefined symbols for architecture x86_64:
"_foo", referenced from:
_main in source.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1
(use -v to see invocation)
```



University of Glasgow

Today's task

- **On moodle there are two examples of the Fibonacci implementation (recursive and iterative) in this folder**

1. **T1: Download them, and run the examples using the remote access system to the university servers.**

– To access the university servers follow the instructions in this word document.

2. **T2: generate all the different representations from preprocessing to object code.**

University of Glasgow

Next week: Basics of C

- Basics of C: variables, `structs`, control flow, functions
- What are data types, why are they important, and what do they tell us about memory?
- If you know C already well you might skip the lecture next week, but do come to the Lab!

University of Glasgow

Systems Programming
Introduction to C (part 2)

Anna Lito Michala

Email AnnaLito.Michala@Glasgow.ac.uk

University of Glasgow

Overview – Introduction to C and Data Types

- Today we are going to discuss fundamental features of C
- We will focus on differences to Java and Python and not discuss every syntactic construct in C
- A particular focus lies on the role *data types* play in assisting us to write *meaningful programs*
- We will discuss important fundamental concepts such as: *lexical scope* and *lifetime* of variables, *call-by-value*, or the difference between *declarations* and *definitions*
- We will also discuss the meaning and importance of compiler warnings and errors
- We will explore all of this by writing example C programs

University of Glasgow

Java vs C

- We start by looking at some `Java` and equivalent `C` programs
- These are taken from rosettacode.org which collects programs in many different languages

1. Computation of the Dot Product: $z = \sum_{i=0}^{n} x_i \cdot y_i$

```java
public class DotProduct {
  public static void main(String[]
args) {
    double[] x = { 1, 3, -5};
    double[] y = { 4, -2, -1};
    double z = 0;
    for (int i = 0; i < 3; i++) {
      z += x[i] * y[i];
    }
    System.out.println(z);
  }
}
```

```c
#include <stdio.h>
int main() {
  double x[] = { 1, 3, -5};
  double y[] = { 4, -2, -1};
  double z = 0;
  for (int i = 0; i < 3; i++) {
    z += x[i] * y[i];
  }
  printf("%f\n", z);
}
```

2. Calculation of the value of e

- The number e is defined as the infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{1} + \frac{1}{1} + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots$$

```java
public class CalculateE {
  public static final double EPSILON =
1.0e-15;
  public static void main(String[] args) {
      long fact = 1;
      double e = 2.0;
      int n = 2;
      double e0;
      do {
        e0 = e;
        fact *= n++;
        e += 1.0 / fact;
      } while (Math.abs(e - e0) >=
EPSILON);
      System.out.printf("e = %.15f\n", e);
  }
}
```

```c
#include <stdio.h>
#include <math.h>
#define EPSILON 1.0e-15
int main() {
  long fact = 1;
  double e = 2.0;
  int n = 2;
  double e0;
  do {
    e0 = e;
    fact *= n++;
    e += 1.0 / fact;
  } while (fabs(e - e0)
>=EPSILON);
  printf("e = %.15f\n", e);
}
```

University of Glasgow

3. Guess the number

- The program picks a random number and let the user guess it

```java
import java.util.Random; import java.util.Scanner;
public class Guess {
  public static void main(String[] args) {
   Scanner scan = new Scanner(System.in);
   Random random = new Random();
   int randomNumber = random.nextInt(10 + 1);
   int guessedNumber = 0;

   do {
    System.out.print("Guess what the number is: ");
    guessedNumber = scan.nextInt();
    if (guessedNumber > randomNumber)
     System.out.println("Your guess is too high!");
    else if (guessedNumber < randomNumber)
      System.out.println("Your guess is too low!");
    else
      System.out.println("You got it!");
   } while (guessedNumber != randomNumber);
  }
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(){
  int number, guess;
  srand( time( 0 ) );
  number = rand() % (10 + 1);
  while( scanf( "%d", &guess ) == 1 ) {
    printf( "Guess what the number is: "
);
    if( number == guess ){
      printf( "You got it!\n" );
      break;
    }
    printf( "Your guess was too %s.\n",
       number < guess ? "high" : "low" );
  }
}
```

University of Glasgow

4. Matrix Transposition

- Transpose a matrix, that is flip it over its diagonal

```java
import java.util.Arrays;
public class Transpose {
  public static void main(String[] args){
    int[][] m = { {1, 1, 1, 1},
                  {2, 4, 8, 16},
                  {3, 9, 27, 81},
                  {4, 16, 64, 256},
                  {5, 25, 125, 625} };
    int[][] ans = new int[4][5];

    for(int row = 0; row < m.length; row++)
      for(int col = 0; col < m[0].length; col++)
        ans[cols][rows] = m[rows][cols];

    for(int[] row : ans) {
      for (int x : row)
        System.out.printf("%3d ", x);
        System.out.println();
    }
  }
}
```

```c
#include <stdio.h>
int main() {
  int m[][] = { {1, 1, 1, 1},
                {2, 4, 8, 16},
                {3, 9, 27, 81},
                {4, 16, 64, 256},
                {5, 25, 125, 625} };
  int ans[4][5];

  for (int row = 0; row < 5; row++)
    for (int col = 0; col < 4; col++)
      ans[col][row] = m[row][col];

  for (int r = 0; r < 4; r++)
    for (int c = 0; c < 5; c++)
      printf("%3d%c", ans[r][c],
        c == 4 ? '\n' : ' ');
}
```

University of Glasgow

Java vs. C

- As seen with the examples Java and C look and feel **very similar** (when we ignore the object oriented features in Java)
- Java has adopted the same syntax style that C introduced earlier

- **If you know Java already you can start writing C code straight away!**

- We will learn the **differences** of how Java and C programs are executed and how memory is organised
- In the rest of this lecture we will look in more depth into C and start exploring some differences with Java

University of Glasgow

Task

Before watching the next video try the following:

- **T1: pick one of the Java examples in this folder**
- **T2: without looking at the C code try to write a C equivalent**
- **T3: check your answer against the C code in the same folder. Note any differences! Was there something that you did not translate correctly? Any counter intuitive parts?**
- **T4: in the zoom session bring your notes! We will discuss them in more detail.**

University of Glasgow

Compiler warnings and errors

- Always remember: **The compiler is your friend!**
- Errors and warnings are feedback from the compiler that there is something wrong with your program
- A compiler error indicates that it is impossible to compile your program. You have to definitely change your program to get it compiling
- A compiler warning indicates an unusual condition that *may* (and quite often do) indicate a problem You should change the program to: *either* fix the problem *or* clarify the code to avoid the warning.
- There is **no good reason to ignore warnings**!
- The `-Werror` flag turns all warnings into errors making it impossible to compile a program with warnings
- The `-Wall` flag enables most compiler warnings
- Every warning and error that is caught by the compiler can not lead to a problem at runtime!
- In this course we insist on using the `-Wall -Werror` flags (particular for your coursework!)

University of Glasgow

Makefile 101

- `make` is the most popular *build system* for C programs automating the process of compiling programs
- Alternative build systems are: Maven, Bazel, Ninja. There exists much more ...
- For make we are writing a Makefile which has the following basic structure:

```
program : source.c
clang -Wall -Werror source.c -o program
# ^ this space must be a single tab character!
```

- The first two lines form a *rule* which explain how a *target* (in our case: program) is build
- The lines below the fist line is the *commands* of the rule and define how the target is build
- After the colon *dependencies* are listed, these are files or other targets which are required to build the target
- Run make will execute the first rule of the `Makefile` in the current directory `make target` will execute the rule to make the named target

University of Glasgow

Entry point for every C program - `main`

- Every C program has exactly *one* `main` function which is the entry point into the program

```c
#include <stdio.h>
int main() {
  printf("Hello world!\n");
}
```

- The `main` function is special in the sense that if the terminal `}` is reached `0` is automatically returned
- A return value of `0` indicates a successful execution to the environment executing the program
- A non-negative return value indicates an unsuccessful execution
- Two valid versions of main:

```c
int main() { ... }
int main(int argc, char* argv[]) { ... }
```

- The second version allows to process command line arguments
  We will learn how to understand the signature next week when we learn about pointers

University of Glasgow

Basic output with `printf`

- `printf` is a function defined in `#include <stdio.h>` and allows a formatted printing of values
- The first argument is the *format string* containing special characters indicating the formatting of values
- Starting from the second arguments are the values to be printed
- The number and order of special characters and values has to match

Special Characters	Explanation	Argument Type
%c	Single character	`char`
%s	Character string	`String: (char *)`
%d	signed integer in decimal representation	`int`
%f	floating-point number in decimal representation	`float double`

Full list of special characters
at: https://en.cppreference.com/w/c/io/fprintf

```
printf("%c %s", 'a', "string"); // prints: "a string"
printf("%f - %f = %d", 3.14, 1.14, 2); // prints: "3.140000 - 1.140000 = 2"
```

University of Glasgow

Variables

- Variables are syntactically used exactly like in Java, e.g.: `int x = 4; x++;`
- Variable *definitions* contain first the *data type*, then the *name* (or *identifier*) and an *initialization expression*

```
int x = 4;
```

- Why do we need to declare the data type for every variable?
- Python doesn't require us to define data types:

```
x = 4
x = x + 1
```

- C is a *statically typed* programming language where every expression has to have a data type which is known without running the program

But what are data types for anyway?

University of Glasgow

Representation of values and Data Types

- What is the *meaning* of the following bit-pattern?

```
1000 0001
```

- Or more precisely, what could this bit-pattern mean?
  - maybe: 129 if we want to represent a unsigned 8-bit long integer value
  - maybe: -127 if we want to represent a signed 8-bit long integer value with two's complement
  - maybe: -1 if we want to represent a 8-bit `Minifloat` value
  - but also maybe it means the color blue? or 129 bananas? ...

- A bit-pattern has no meaning on it's own! We, the human programmers, give bit-patterns meaning!
- By declaring a variable with a *data type* (such as int) we decide what the bit-pattern in memory *means*

Data types give bits meaning!
(That's why they are so important)

University of Glasgow

Preserving meaning thanks to data types

- To assist us writing *meaningful programs* the compiler enforces that computations *preserve the meaningful representation of our data*
- For example: for x+1 the compiler ensures that a *meaningful* addition of the value one and x is performed.

When x has the data type char and the value 42 this will modify the bit-pattern like this:

```
0010 1010 + 0000 0001 = 0010 1011
```

When x has the data type float and the value 42 this will modify the bit-pattern like this:

```
0 10000100 01010000000000000000000
+ 0 01111111
00000000000000000000000
=================================
=
0 10000100 01011000000000000000000
```

- By enforcing operations to respect the data types the compiler prevents meaningless computations:

```
error: invalid operands to binary + (have ‘float’ and ‘char
*’)
x = x + "1";
      ^
```

University of Glasgow

Representation of C variables in memory

- Every variable in C is stored at a constant location in memory that does not change over its lifetime
- In C the data type of a variable determines its representation in memory:

```
int x = 42;   // represented as: 0000 0000 0000 0000 0000 0000 0010
1010
float f = 42; // represented as: 0100 0010 0010 1000 0000 0000 0000
0000
```

- By choosing a particular integer data type we are in control how much memory we use!

Type name	Typical size in bytes	Value range
char / unsigned char	1 byte	[-127, +127] / [0, 255]
short / unsigned short	2 bytes	[-32767, +32767] / [0, 65535]
int / unsigned int	4 bytes	[-2147483647, +2147483647] / [0, 4294967295]
long / unsigned long	4 or 8 bytes	at least as for `int`
long long / unsigned long long	8 bytes	$[-2^{63}-1, +2^{63}-1]/[0, 2^{64}-1]$

- For example to store 1 million temperature measurements in Celsius choosing an array of `char` values over an array of int values safes almost 3 MB of data this is almost the entire size of my processors L3 cache!

University of Glasgow

Boolean values in C

- We have now seen multiple data types for representing integer and floating point values
- What about the boolean values `true` and `false`?
- In C every integer can be interpreted as a boolean, where `0` represents `false` and any other value `true`:

```
int i = 5;
while (i) {
    i = i - 1;
    printf("%d\n", i);
}
printf("done\n");
```

prints

```
4
3
2
1
0
done
```

- Since the C99 standard it is also possible to use the `#inlude <stdbool.h>` header:

```
#inlude <stdbool.h>
int main() { bool a = true; bool b = false; bool c = a && b; printf("%d\n",
c); }
```

- In `stdbool.h` `true` and `false` are declared as macros with the values `1` and `0`

University of Glasgow

Task

- **T1: Check out the Makefile in this folder**
- **T2: Use the command line `make` and see if your program compiles!**
- **Makefiles can be as simple or as complicated as you need them to be. You can read more on them here if you are interested, but it is not mandatory!**
- **Remember to bring any issues, observations or questions with you at the Zoom meeting!**

University of Glasgow

Lexical Scoping

- Each pair of curly braces { } is called a block in C and introduces a *lexical scope*
- Variable names must be unique in the same lexical scope
- For multiple variables with the same name, the variable declared in the innermost scope is used.

Which values will be printed?

```c
int main() {
  int i = 5;
  {
    int j = foo(i);
    printf("%d\n", j);
  }
}
```

```c
int foo(int i) {
  int j = i;
  {
    int j = i + 2;
    printf("%d\n", j);
  }
  return j + 1;
}
```

- First 7 (from inside foo) and then 6 (from inside main) will be printed

University of Glasgow

Scoping and preprocessor macros

- Preprocessor macros are dangerous as they don't respect the lexical scoping
- Consider the following macro

```
#define ADD_A(x) x + a
```

- We could not define a function like this, because a is not in the lexical scope of the definition
- If we now apply this macro it will work with what ever a is in scope where we *use* the macro:

```
int add_one(int x) {
  int a = 1;
  return ADD_A(x); // will expand to: return x + a; and compute x + 1
}
int add_two(int x) {
  int a = 2;
  return ADD_A(x); // will expand to: return x + a; and compute x + 2
}
```

- This is called *dynamic scoping* (in contrast to lexical scoping) and should best be avoided!

University of Glasgow

Variable lifetime

- We have learned that variables are stored at locations in memory that do not change over their lifetimes
- But what is the *lifetime* of a variable? In C this depends how the memory for the variable was allocated. There are three cases:

1. *automatic*: These are all variables declared locally in a block (i.e. inside a pair of {}) and their lifetime ends at the end of the block. All variables we have seen so far fall into this category.

```
int main() {
  int x = 42;
} // end of the block - end of the lifetime of x
```

2. *static*: Variables declared with the static keyword or defined at file-level outside all blocks. The lifetime is the entire execution of the program.

```
int foo() {
  static int count_calls_to_foo = 0; // the variable is initialized only once
  count_calls_to_foo++;
} // variable continues to life
```

3. *allocated*: These are variables for which we have to explicitly request memory using dynamic memory allocation functions (such as malloc). We manage the lifetime of these variables ourselves.

University of Glasgow

Stack-based Memory management

- When the lifetime of an ***automatically*** managed variable ends, its memory location is freed and can be reused by other variables
- This memory management happens fully automatically as it is very easy to implement:
  - every time a block is entered put aside a location in memory for every variable declared in the block
  - every time a block is exited free the locations in memory for every variable declared in the block
- As this memory management strategy addes and removes memory locations in a *last-in-first-out* manner we call this *stack-based memory management* and the area of memory managed in this way *the* ***stack***

- We will learn in the next two weeks the other important area of memory called the ***heap*** which is manged *manually* by the programmer

Types combining multiple elements -Struct

- So far we have only seen variables with basic types.
  There are two important types which combine multiple elements: `arrays` and `structs`
- A `struct` consists of a sequence of *members* of (potentially) different types:

```
struct point {
  int x;
  int y;
};
int main() {
  struct point p = {1, 2};
  printf("x = %d\ny = %d\n", p.x,
p.y);
}
```

- The members are accessed like public class members in Java using the . notation
- The members are stored next to each other in memory in the same order as defined in the struct
- The type of a struct is written `struct name`, but we often use this trick to shorten it:

```
typedef struct { int x; int y; } point; int main() { point p = {1, 2}; /* ...
*/ }
```

University of Glasgow

Types combining multiple elements -Array

- Arrays consist of a multiple elements of the same type:

```
int main() {
  int x[2] = {1, 2};
  printf("x[0] = %d\nx[1] = %d\n", x[0], x[1]);
}
```

- Arrays which are stored on the stack must have a fixed size (so that the memory is automatically managed)
- We will learn how to use arrays for which the size can change next week
- The elements on an array are stored next to each other in memory

Types combining multiple elements -String

- Strings in **C** are represented as arrays of characters

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

is the same as

```
char greeting[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\0'};
```

- We use double quotes " to write a (ASCII) string literal and single quotes ' to write a character literal
- Strings in C are terminated by the special character '\0' that is added automatically for string literals
- To print a string with printf we use the %s formation character

```
printf("%s\n", greeting);
```

- If we forget the terminating `'\0'` character this will print the content of the memory until it hits the next bit pattern equivalent of `'\0'`!

University of Glasgow

Functions -Definition

- Functions are probably *the most important* abstraction mechanism in computing science They allow us to write code in a modular fashion which we can then reuse
- A *function definition* in C looks like this:

```
int max(int lhs, int rhs) {
if (lhs > rhs) { return lhs; } else { return rhs; }
}
```

- The *return type* specifies the data type of the value that will be returned after evaluating the function. If a function returns no value the special type `void` is used as return type
- The *name* which should describe the behaviour of the function
- A *parameter list* which specifies the data type and name of each parameter expected by the function
- The *function body* which is a block containing the code executed when calling the function
- To call a function we provide an *argument* for each parameter and process the return value

```
int x = max(5, 3);
```

University of Glasgow

Functions -Declaration vs. Definition

- A function definition (previous slide) fully specifies the behaviour of the function
- A *function declaration* only specifies the interface describing how a function can be used:

```
int max(int lhs, int rhs);
```

- Function declarations are important for writing modular software as it allows to separate the interface (the declaration) from the implementation (the definition)
- For calling a function (e.g. `max` or `printf`) the compiler checks that the data types of the call expression and the function declaration match:

```
int max(int lhs, int rhs);
int main() { int x = max(4.5, 'b'); }
```

```
max.c:6:17: error: implicit conversion from 'double' to 'int' changes value from 4.5 to 4 [-Werror,-Wliteral-conversion]
int x = max(4.5, 'b');
        ~~~ ^~~
```

- The linker searches for the definition which might be in a different file or library (as for `printf`)

University of Glasgow

Functions - Call-by-value

- When we call a function how are the arguments passed to it?
  E.g. what happens to a variable which is passed to a function which modifies its parameters inside its body?

```
void set_to_zero(int x) { x = 0; }
int main() {
int y = 42;
set_to_zero(y);
printf("%d\n", y); // what will be printed?
}
```

- The value 42 will be printed, because when we call a function we pass *a copy of the arguments* to it
- That means x and y in the example above are stored at different locations in the memory
  When set_to_zero is called the value of y is copied into the memory location of x
- We call this *call-by-value* as the argument value is evaluated and stored in the parameter variable

```
void foo(int x)
int main() { foo( 5 + 4 ); /* first evaluate 5 + 4 to 9 and then store it in x */ }
```

University of Glasgow

Functions - Call-by-value - Structs and Arrays

- Functions can accept `structs` and `arrays` as parameters:

```
typedef struct { int x; int y; } point;

int search(point needle, point haystack[], int haystack_size) {
  for (int i = 0; i < haystack_size; i++) {
    point candidate = haystack[i];
    if (needle.x == candidate.x && needle.y == candidate.y) { return i; }
  }
  return -1;
}
```

- `structs` behave like the basic types and are passed-by-value
  Changes to the `struct` 's members are therefore not visible outside the function
- `arrays` are treated slightly specially:
  - instead of copying the array (which would be expensive) the *address of the first element* is passed
  - because of this changes to the `array` elements **are** now visible outside the function
- The special treatment of `arrays` will make more sense when we learn about *pointers* next week

University of Glasgow

Task

- **Before the zoom session today**

– T1: give all the Example in this folder programs a try

– T2: Try to implement a struct and using it as a parameter for a function (see previous slide)

– T3: Alter the struct element values in the function and print the struct after the function call. See if your changes were persistent.

– T4: bring any observations or questions with you at the zoom meeting!

University of Glasgow

Next week

- Next week we are going to deepen our understanding of memory
- We will learn what *pointers* are and why they are fundamental in understanding C and memory
- We will learn the first steps of using dynamic memory allocation on the heap using malloc

University of Glasgow

Systems Programming
Memory & Pointers

Anna Lito Michala

Email AnnaLito.Michala@Glasgow.ac.uk

What is (computer) memory?

From the Oxford English Dictionary:

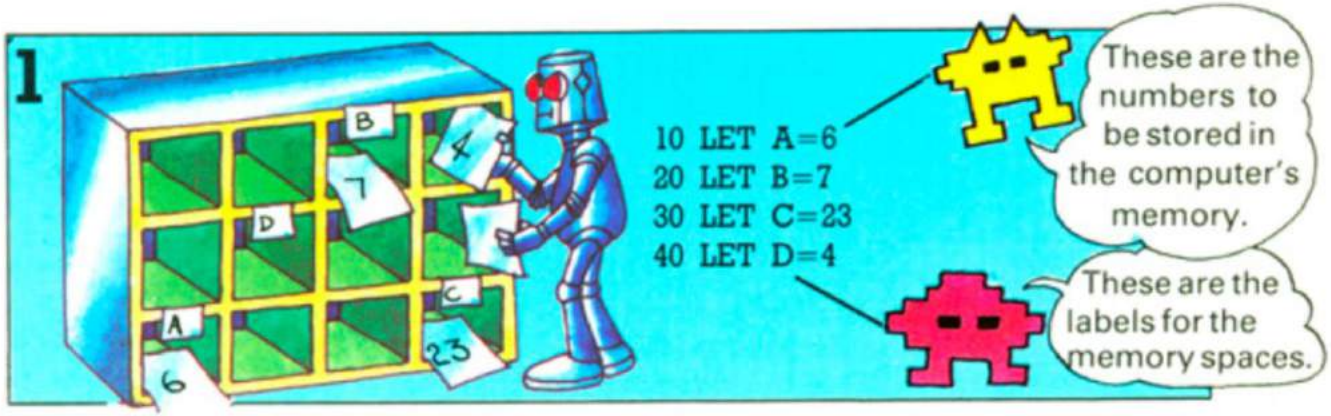
memory | ˈmɛm(ə)ri |

noun

1. the faculty by which the mind stores and remembers information
2. something remembered from the past
3. *the part of a computer in which data or program instructions can be stored for retrieval.*

How should we think about memory?

Introduction to Computer Programing
(Brian Reffin Smith, (Usborne, 1982))



When you put a piece of data into the computer's memory you have to give it a label so you can find it again. You can use letters of the alphabet as labels. To label a memory space and put a number in it you can use the word LET, as shown above. A labelled memory space is called a variable because it can hold different data at different times in the program.

We can think of memory as a sorting cabinet where each box stores the value of a variable
The variable name is the *label* which allows us to remember where we stored what

University of Glasgow

How should we think about memory?

- We can also think of memory as a single long street (called *memory lane*) where each house has a **unique *address***
- We have some notion of *spacial* *locality*: Houses close to each other are neighbours, but there are also houses far away at the other end of memory lane

  Spacial locality is an important property exploited by *caches* (separated fast but small memories)

University of Glasgow

Computer Memory in Programming

- We can think of memory as a very large one-dimensional array (or a large buffer) where instead of indices into the array we use names to identify variables
- These two programs behave exactly the same:

```c
int main() {

  int x = 42;
  int y = 23;
  int z = x * y;
}
```

```c
int main() {
  int memory[3];
  memory[0] = 42;
  memory[1] = 23;
  memory[2] = memory[0] * memory[1];
}
```

- Obviously, arrays only store elements of the same type where memory stores variables of different types

University of Glasgow

Byte Addressable Memory

- Every *byte* (a collection of 8 bits) in memory has it's own *address*
- On a 64-bit architecture these addresses are 64-bit (or 8 bytes) long
- That means a 64-bit architecture can address (in theory) up to 264264 bytes = 16 exabytes
- In practice x86-64 only uses the lower 48 bits of an address, supporting up to 248248 bytes = 256 TB

- To modify a single bit we always have to load and store an entire byte:

```
int main() {
  unsigned char byte = 0;
  byte = byte | (1 << 3); // set the 3rd bit to 1
  printf("%d\n", byte); // prints 8 (== 2^3)
}
```

University of Glasgow

Variables in memory

- As we have learned last week:
  every variable in C is stored at a constant location in memory that does not change over its lifetime
- The location of a variable identifiable by it's *address*
- Depending on the size of the data type the value of the variables will span multiple bytes in memory
- We can ask the address of a variable in C using the ***address-of* operator &**:

```c
int main() {
  int x = 42;
  int y = 23;
  printf("&x = %p\n", &x); // print the address of x
  printf("&y = %p\n", &y); // print the address of y
}
```

- We can ask the size of a variable using the `sizeof` operator:
  In the example `sizeof(x)` prints `4` as an int value is represented by 4 bytes

University of Glasgow

Pointers

- We can store the address of a variable as the value of *another variable* that we call a ***pointer***

```
int main() {
  int x = 42;
  int * pointer_to_x = &x; // this is a pointer referring to x
  printf("value of pointer_to_x: %p\n", pointer_to_x); // prints 0x77...
```

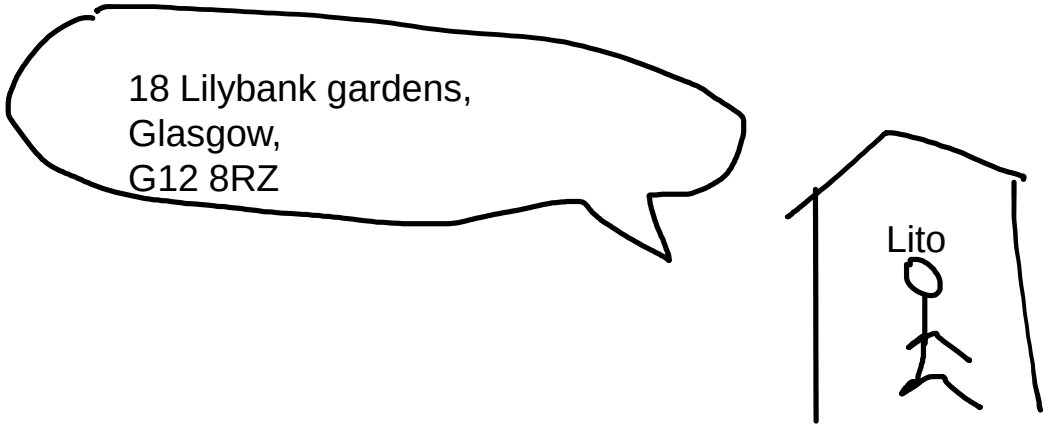
- The ***dereference operator*** ***** allows us to access the value of the variable we are pointing to:

```
  printf("value of x: %d\n", *pointer_to_x); // prints 42
}
```

- A pointer to a variable of data type `t` has the data type `t *`
- Every pointer has the same size (independent of the type it's pointing to) because it stores an address
- Pointers on a 64-bit architecture are 8 bytes (or 64 bits)

Back to the houses example

- **Variable: your house**
- **Pointer: your home address**
- **Value of the variable: you inside your house**




```
  printf("value of pointer_to_x: %p\n", pointer_to_house); // prints 18
Lilybank...
  printf("value of x: %d\n", *pointer_to_house); // prints Lito
}                    //     ~~~^~~~
```

University of Glasgow

Pointers are normal variables

- A pointer is a variable like any other, that means:
- The pointer is stored at its own location

```
int x = 42; // stored at 0x7ffeedbed3dc
int * ptr = &x; // stored at 0x7ffeedbed3d0
```

- We can take the address of where the pointer is stored using the address operator:

```
printf("%p\n", &ptr); // prints 0x7ffeedbed3d0
```

- We can store the address of a pointer in another pointer:

```
int * * ptr_to_ptr = &ptr; // stored at 0x7ffeed7ed3c8
```

- We can change were a pointer points to:

```
int y = 23; // stored at 0x7ffeebaf23c4
ptr = &y;
```

University of Glasgow

Pointers and Null

- Sometimes there is no meaningful value for a pointer at a certain time
- We use the value `0` or the macro `NULL` to represent a pointer that points to *nothing*
- NULL often represents an erroneous state, e.g. that an element wasn't found in an array

```
// return pointer to value found in array
// return NULL if value not found in array
float* search(float needle, float haystack[], int haystack_size) {
  for (int i = 0; i < haystack_size; i++)
  if (needle == haystack[i]) { return &haystack[i]; }
  return NULL;
}
```

- Dereferencing a NULL pointer will crash your program!
  This has led to many software bugs and their inventor Tony Hoare has called it his *billion-dollar mistake*

University of Glasgow

Pointers and `const`

- In C every variable can be annotated with the *type qualifier* `const` to indicate that the content of the variable can not be changed
- This is enforced by the compiler:

```
  const_error.c:4:6: error: cannot assign to variable 'pi' with const-qualified type 'const
float'
  pi = 2.5;
  ~~ ^
const_error.c:3:15: note: variable 'pi' declared const here
  const float pi = 3.14;
  ~~~~~~~~~~~~~^~~~~~~~
```

- Pointers can be const, i.e. unmodifiable, in three ways:

1. The *pointer itself*, i.e. the address, can not be changed: `float * const ptr;`
2. The *value we are pointing to* can not be changed: `const float * ptr;`
3. *Both* value and pointer can not be changed: `const float * const ptr;`

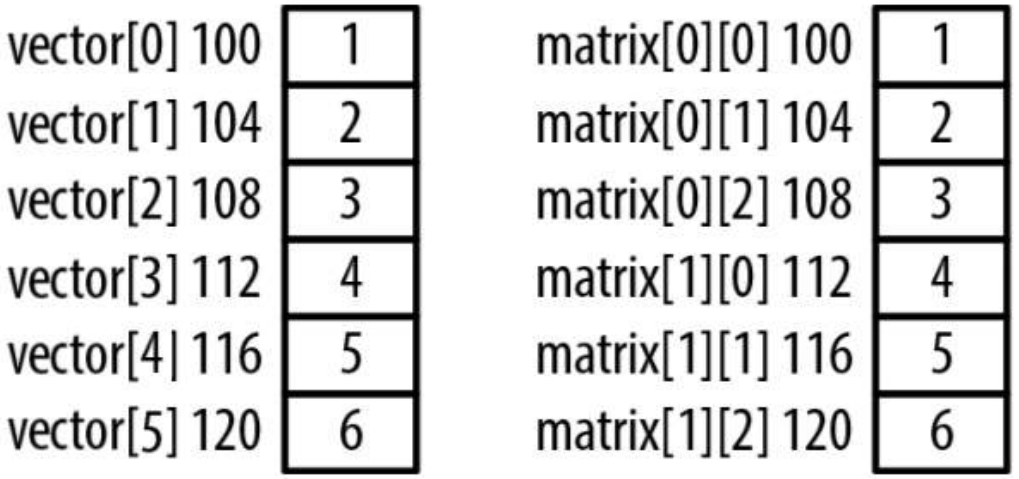
University of Glasgow

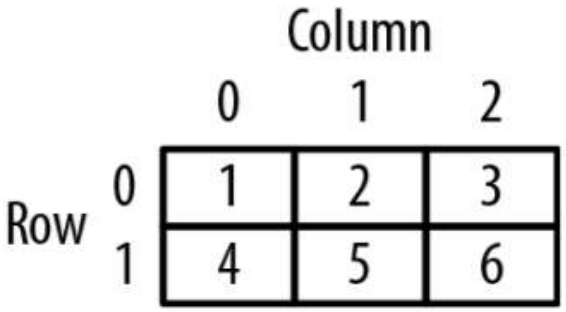
Arrays in Memory

- Last week we have seen one- and two-dimensional arrays:

```
int vector[6] = {1, 2, 3, 4, 5, 6};
int matrix[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

- vector and matrix in the example are both stored exactly the same in memory:





- The two-dimensional array is stored in a *row major* format (i.e. rows are stored after another)

University of Glasgow

Pointers and Arrays

- Pointers and arrays are closely related
- The name of an array refers to the *address of the first element* when assigned to a pointer variable:

```
int vector[6] = {1, 2, 3, 4, 5, 6};
int * ptr = vector; // this is equivalent to: int * ptr = &(vector[0]);
```

- We can use the array indexing notation on pointers:

```
printf("5th element: %d\n", ptr[4]); // prints "5th element: 5"
```

- Two important differences:

– `sizeof` returns different values (size of array vs. size of pointer):

```
printf("%ld\n", sizeof(vector)); // prints '24' (== 6 * 4 bytes)
printf("%ld\n", sizeof(ptr)); // prints '8' (size of a pointer)
```

– we can not change a vector, only its elements:

```
vector = another_vector; // error: array type 'int [6]' is not assignable
```

University of Glasgow

Pointer Arithmetic

- We can use *pointer arithmetic* to modify the value of a pointer. We can:

1. Add and subtract integer values to/from a pointer
2. subtract two pointers from each other
3. compare pointers

```c
int vector[6] = {1, 2, 3, 4, 5, 6};

int * ptr = vector; // start at the beginning
while (ptr <= &(vector[5])) {
  printf("%d ", *ptr); // print the element in the array
  ptr++; // go to the next element
}
```

- Pointer arithmetic takes into account the size of the type the pointer is pointing to:

```c
int * i_ptr = &i; char* c_ptr = &c;
i++; // this adds 4-bytes (sizeof(int)) to the address stored at i_ptr
c += 2; // this adds 2-bytes (2 * sizeof(char)) to the address stored at
c_ptr
```

- The expressions `ptr[i]` and `*(ptr + i)` are equivalent

University of Glasgow

Linked List with pointers and `structs`

- Pointers and structs are useful in building data structures, as example we look at a (single) linked list
- A linked list contains of *nodes* with a *value* and a *pointer to the next node*:

```
struct node { char value; struct node * next; };
```

- The last node in the list has a next-pointer to NULL

```
int main() {
  struct node c = {'c', NULL};
  struct node b = {'b', &c};
  struct node a = {'a', &b};
```

- We use a pointer to iterate over the linked list:

```
  struct node * ptr = &a;
  while (ptr) {
    printf("%d\n", (*ptr).value);
    ptr = (*ptr).next;
  }
}
```

University of Glasgow

Binary (Search) Tree

```
struct node {
  char value;
  struct node * left_child;
  struct node * right_child;
};
```

- If we keep the binary tree ordered (thus it forms a *binary search tree*) we can search efficiently in it:

```
node * find(char value , node * root) {
if (value == root->value) {
return root;
}
if (value < root->value && root->left_child != NULL)
{
return find(value, root->left_child);
}
if (value > root->value && root->right_child !=
NULL) {
return find(value, root->right_child);
}
return NULL;
}
```

- `s_ptr->m` is a notation to access member `m` of a struct-pointer `s_ptr` and is equivalent to `(*s_ptr).m`

University of Glasgow

Call-by-value revisited

- We learned last week that arguments are passed *by-value* to functions
- That means that the value of the argument is copied into the parameter variable
- This is also true for pointers
- Arrays are treated specially and a *pointer to the fist element* is copied instead of the entire array

```
float average(float array[], int size) {
  float sum = 0.0f;
  for (int i = 0; i < size; i++) { sum += array[i]; }
  return sum / size;
}
```

- The array is treated like a pointer, in fact `int param[]` and `int * param` are interchangeable:

```
float average(float array[], int size);
float average(float * array, int size);
```

University of Glasgow

`main` with command line argument

- This allows us to understand the definition of the main function that processes command line arguments

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

- The `argc` argument specifies the number of command line arguments
- The `argv` argument specifies an array of the command line arguments as strings
  - A single string is represented as an array of characters: `char *`
  - The type of `argv char * []` can also be written `char * *`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
  // print every command line argument
  for (int i = 0; i < argc; i++)
  printf("%s\n", argv[i]);
}
```

University of Glasgow

Writing Generic Code with `void *`

- Sometimes we want to write code in a generic way so that it works will all data types e.g. swapping two variables or sorting algorithms
- To swap two variables x and y of arbitrary type we copy all bytes at the location of x to y and vise-versa
- For this we write a function `swap` that takes two pointers and number of bytes that have to be swapped:

```
void swap(void *x, void *y, size_t l)
{
  char *a = x, *b = y, tmp;
  while(l--) {
    tmp = *a;
    *a++ = *b;
    *b++ = tmp; }
}
```

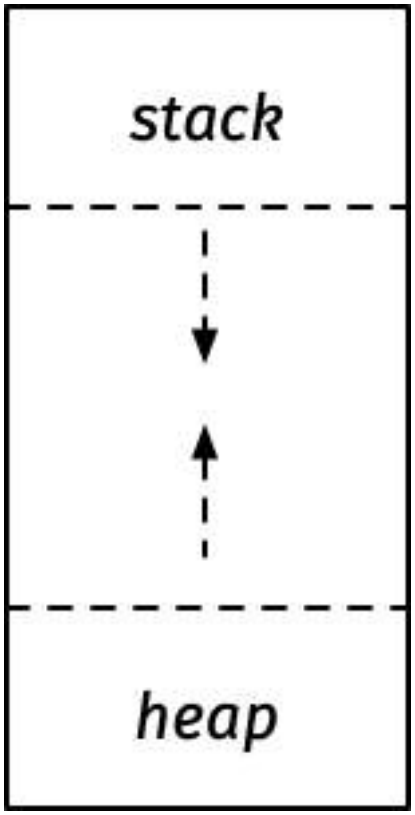
- The special pointer of type `void *` is a **generic pointer**. Every pointer is automatically convertible to it

A `void`-pointer only serves as an address pointing to *something*.
We can not access the value we are pointing to as we don't know what those bits mean.
Dereferencing a `void` pointer is forbidden.

University of Glasgow

Stack vs. Heap as memory regions

- So far we have only seen variables with *automatic* lifetime managed by the compiler
- These variables are stored in a part of memory that is called the ***stack***
- The size of every variable on the stack has to be known statically, i.e. without executing the program
- For many important use cases we don't know the size of a variable statically, e.g. dynamically sized arrays
- For such cases we manage the memory manually by dynamically requesting and freeing memory
- The part of the memory used for dynamic memory allocation is called the ***heap***
- Stack and heap share the same address space and grow with use towards each other



stack and heap are in a single address space

University of Glasgow

Dynamic memory management

- We request a new chunk of memory from the heap using the `malloc` function:

```
void* malloc( size_t size ); // defined in <stdlib.h>
```

- We specify the number of bytes we like to allocate
- If `malloc` succeeds a void-pointer to the first byte of the un-initialised memory is returned
- If `malloc` fails a `NULL` pointer is returned

- Memory allocated with malloc must be manually deallocated by calling free (exactly once):

```
void free( void* ptr ); // defined in <stdlib.h>
```

- If `free` is not called we *leak* the allocated memory

University of Glasgow

Dynamically sized array example

- We use `malloc` to implement dynamically sized `arrays`
- Here the size of the `array` depends on the first number entered by the user

```c
#include <stdlib.h>
#include <stdio.h>
int main() {
  printf("How many numbers do you want to average?\n");
  int count;
  if (scanf("%d", &count) == EOF) { exit(-1); }
  // allocate memory based on dynamic input (here its size)
  int* array = malloc(count * sizeof(int));
  for (int i = 0; i < count; i++) {
    int number;
    if (scanf("%d", &number) == EOF) { exit(-1); }
    array[i] = number;
  }
  float sum = 0.0f;
  for (int i = 0; i < count; i++) { sum += array[i]; }
  printf("The average is %.2f\n", sum / count);
  free(array); // free the memory manually after use
}
```

University of Glasgow

Linked list extended example

- Using `malloc` we are now able to define functions for creating, extending, and freeing lists:

```
struct node { char value; struct node * next; }; // same as before
struct node* create_node(char value) {
  struct node * node_ptr = malloc(sizeof(struct node));
  node_ptr->value = value; node_ptr->next = NULL;
  return node_ptr;
}
void add_node_to_list(struct node* list, struct node* node) {
  if (!list) return;
  while (list->next) { list = list->next; }
  list->next = node;
}
void free_list(struct node* list) {
  if (!list) return;
  while (list->next) {
    struct node* head = list;
    list = list->next;
    free(head);
  }
}
```

University of Glasgow

Returning a pointer to a local variable

- It is an easy mistake to return a pointer to a local variable. Never do it!
- Why?
- By returning a pointer to a local variable the pointer has a longer lifetime than the variable its pointing to

```
struct node* create_node(char value) {
struct node node;
  node.value = value; node.next = NULL;
  return &node;
} // lifetime of node ends here ... but its address lives on in
a_ptr

int main() {
  struct node* a_ptr = create_node('a');
  // ...
} // lifetime of a_ptr (pointing to node) ends here
```

Never return a pointer to a local variable from a function!

University of Glasgow

Function Pointers

- Memory does not store data but also program code. It is possible to have a pointer pointing to code!
- These pointers are called *function pointers* and have the type:

```
return_type (*)(argument_types)
```

- A common use case is passing functions as arguments to other functions:

```
// defined in <stdlib.h>
void qsort( void *ptr, size_t count, size_t size,
  int (*comp)(const void *, const void *) );
// defined in qsort_example.c
int compare(void* fst, void* snd) { /* ... */ };
int main() {
  // ...
  qsort(array, array_length, sizeof(array[0]),
compare);
  // ...
}
```

- Here `compare` is passed as argument to `qsort` and assigned to the function pointer `comp` in line 3
- Function names are automatically converted to function pointers, so that we do not have to write `&compare`

University of Glasgow

Example – Generic Quicksort

- `qsort` in C is written generically so that it can sort data of any data type
- `compare` takes two `void`-pointers and returns a negative integer if the first argument is *less than* the second, a positive integer if it is *greater than* and zero if they are *equal*
- `compare` has to ***cast*** (i.e. convert) the pointers into a concrete pointer type before dereferencing them:

```
int compare(const void* fst, const void* snd) {
  const float* f_fst = fst; const float * f_snd =
snd;
  return *f_fst - *f_snd;
}
```

- This is a good example of the use of a `void`-pointer:
  when passing the pointers `fst` and `snd` around in `qsort` these are addresses pointing *somewhere*
  inside of `compare` we first have to give meaning to the bits we are pointing to before we are able to make a meaningful comparison

University of Glasgow

Next week

- We are going to look at **resource management** and **ownership**
- We will particular focus on **memory management**
- We will learn a technique in **C++** that assists ownership-based resource management

University of Glasgow

Task

To practice some of what we learned today try to write this very short program:

- **T1: ask the user to pass in an integer as command line argument**
- **T2: use this integer to create a dynamically allocated array**
- **T3: add values 0 to size-1 as elements in the newly formed array**
- **T4: for every element of the array print the element's address (pointer) and the element's value**

  » Eg of first output line: 000000ff 0
- **T5: change the for loop so that you use pointer arithmetic and see if your output is the same!**

Bring your findings and/or questions to the zoom meeting!

University of Glasgow

Systems Programming
Memory Management & Ownership

Anna Lito Michala

Email AnnaLito.Michala@Glasgow.ac.uk

University of Glasgow

Recap from last week: Memory

- Every *byte* (a collection of 8 bits) in memory has it's own (usually) 64-bit long *address*
- Every variable in C is stored at a constant location in memory that does not change over its lifetime
- The location of a variable identifiable by it's *address*
- We can ask the address of a variable in C using the ***address-of* operator &**
- We can ask the size of a variable using the `sizeof` operator

```c
int main() {
  int x = 42;
  int y = 23;
  printf("&x = %p\n", &x); // print the address of x
  printf("&y = %p\n", &y); // print the address of y
}
```

University of Glasgow

Recap from last week: Pointers

- We can store the address of a variable as the value of *another variable* that we call a *pointer*
- A pointer is a variable like any other (stored at its own location)
- The ***dereference operator*** ***** allows us to access the value of the variable we are pointing to
- A pointer to a variable of data type `t` has the data type `t *`
- The special pointer of type `void *` is a generic pointer. *Dereferencing a void pointer is forbidden.*
- We use the value `0` or the macro `NULL` to represent a pointer that points to *nothing*
- Dereferencing a `NULL` pointer will crash your program!

```
int main() {
  int x = 42;
  int * pointer_to_x = &x; // this is a pointer referring to x
  printf("value of pointer_to_x: %p\n", pointer_to_x); // prints
0x77...
```

University of Glasgow

Dynamic memory management in C

- We request a new chunk of memory from the heap using the `malloc` function:

```
void* malloc( size_t size ); // defined in <stdlib.h>
```

- We specify the number of bytes we like to allocate
- If `malloc` succeeds a void-pointer to the first byte of the un-initialised memory is returned
- If `malloc` fails a `NULL` pointer is returned

- Memory allocated with malloc must be manually deallocated by calling free (exactly once):

```
void free( void* ptr ); // defined in <stdlib.h>
```

- If `free` is not called we *leak* the allocated memory

University of Glasgow

Binary Tree

- Using `malloc` and `free` we can build data structures like binary trees:

```
struct node { const char * value; struct node * left_child; struct node * right_child; };
typedef struct node node;
node * create_tree(const void * value, node * left_child, node * right_child) {
  node * root = malloc(sizeof(node));
  if (root) {
    root->value = value;
    root->left_child = left_child;
    root->right_child = right_child; }
  return root;
}
void destroy_tree(node * root) {
if (root) {
  destroy_tree(root->left_child);
  destroy_tree(root->right_child);
  free( root ); }
}
int main() {
  node * root = create_tree("b",
  create_tree("a", NULL, NULL),
  create_tree("c",
  create_tree("d", NULL, NULL),
  create_tree("e", NULL, NULL)
    )
  );
}
```

University of Glasgow

Generic Binary Tree

- How can we write a generic binary tree that stores a value of arbitrary type in each node?
- *Solution:* use the generic `void *` to store an address in the node.
  We don't care what data is stored at the address.

```
struct node { const void * value_ptr; struct node * left_child; struct node *
right_child; };
```

- Challenge: how do we write a print function that prints each value?
  Remember, we are not allowed to dereference any `void *` to access its value.

```
void print(node * root) {
  if (root) {
    // ... how to print the value at: root->value_ptr ?
    print(root->left_child);
    print(root->right_child);
  }
}
```

- We need a way to allow the caller of this function to specify how to print the value.

University of Glasgow

Function Pointers

- Memory does not store data but also program code. It is possible to have a pointer pointing to code!
- These pointers are called *function pointers* and have the type: `return_type (*)(argument_types)`
- Now we can implement the print function:

```
void print_string(const void * value_ptr) {
  char * string = value_ptr; // by changing the type we give the bits again
meaning
  printf("%s\n", string);
}
void print(node * root, void (* print_function)(const void *) ) {
  if (root) {
    print_function(root->value_ptr);
    print(root->left_child, print_function);
    print(root->right_child, print_function); }
}
int main() {
  node * root = ... ;
  print(root, print_string);
}
```

- Function names are automatically converted to function pointers, we don't have to write `&print_string`

University of Glasgow

Example: Quicksort

- In this example compare is passed as argument to `qsort` and assigned to the function pointer `comp` in line 3

```
// defined in <stdlib.h>
void qsort( void *ptr, size_t count, size_t size,
int (*comp)(const void *, const void *) );
int main() {
  // ...
  qsort(array, array_length, sizeof(array[0]), compare);
  // ...
}
```

- `compare` takes two `void`-pointers and returns a negative integer if the first argument is *less than* the second, a positive integer if it is *greater than* and zero if they are equal
- `compare` has to *cast* (i.e. convert) the pointers into a concrete pointer type before dereferencing them:

```
int compare(const void* fst, const void* snd) {
  const float* f_fst = fst; const float * f_snd = snd;
  return *f_fst - *f_snd;
}
```

University of Glasgow

Memory Management Challenges

- When we allocate memory ourself with `malloc` we are responsible for calling `free`
- We must call `free` *exactly* once for each address we obtained from `malloc`
- It is good practice to assign the `NULL` value to pointers that have been `freed`
  But this does not prevent all *double free* errors:

```
int main() {
  void * ptr1 = malloc(sizeof(int));
  void * ptr2 = ptr1;
  free(ptr1);
  ptr1 = NULL;
  free(ptr1); // OK. Calling free with NULL is fine
  free(ptr2); // *** error for object 0x7f9355c00690: pointer being freed was not
allocated
}
```

- Another problem are *dangling pointers* which point to locations which have been freed:

```
int main() {
  node * left_child = create_tree("a", NULL, NULL);
  node * root = create_tree("b", left_child, NULL );
  destroy_tree(left_child); // now: root->left_child points to freed memory!
}
```

University of Glasgow

More Memory Management Challenges

- If we never call `free` for an heap-allocated pointer we call this a ***memory leak***

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

int main() {
  char *mem = (char*) malloc( sizeof(char) * 20); // allocate some memory

  if (!mem) { exit(EXIT_FAILURE); } // check if allocation went fine

  // use allocated memory
  mem = (char*) malloc( sizeof(char) * 30); // allocate more memory.
  // we just lost the pointer to the old memory
  // use newly allocated memory

  free(mem); // free newly allocated memory
  // we leaked the memory allocated first
}
```

University of Glasgow

Ownership

- To organise memory management we adopt the concept of *ownership*
- *Ownership* means that we identify a single entity which is responsible for managing a location in memory
- For example we might say that the parent in a tree *owns* its children
  The parent is responsible for allocating and freeing the memory used by the children.
- The C language does not assist in this as it is not enforcing the ownership model
- Next we will learn a feature in C++ that does help us to follow the ownership model
- In rust ownership is enforced by the compiler which makes *double free* and *dangling pointers* impossible and prevents most common cases of leaking of memory

University of Glasgow

Ownership in C++: RAII

- In C++ we can express ownership of a location of heap memory explicitly in the code
- The creator of C++ has invented a programming idiom for this: RAII - *Resource acquisition is initialization*
- The idea is that:

1. we tie the management of a resource to the lifetime of a variable on the stack.
2. The allocation of the resource (e.g. the malloc call) is done when we create the variable.
3. The deallocation of the resource (e.g. the free call) is done when the variable is destroyed.

```
int main() {
  std::vector<int> v = {1, 2, 3, 4, 5, 6}; // heap memory is allocated for the numbers
  // use v
} // memory on the heap will be freed automatically
```

- In **C++** this idiom is implemented by `struct` data types with two *member functions* (called methods in Java):

– The *constructor* which is called when creating values

– The *destructor* which is called when a variable of this type reaches its lifetime

Implementing RAII in C++ - Example

- If we want to store a number of integer values on the heap, we create a special `struct` data type:

```
struct ints_on_the_heap { // This is C++ code and not legal C!
  int * ptr;
  // constructor
  ints_on_the_heap(int size) {
    ptr = (int*)malloc(sizeof(int) * size);
  }
  // destructor
  ~ints_on_the_heap() { free(ptr); }
};
typedef struct ints_on_the_heap ints_on_the_heap;
```

- Initializing a variable on the stack with this data type allocates memory on the heap
- Once the variable on the stack reaches it lifetime it will automatically free the memory on the heap

```
int main() {
ints_on_the_heap i(23); // i is on the stack; memory for 23 ints is allocated on
the heap
i.ptr[22] = 42;
} // automatic call to ~ints_on_the_heap will free heap memory
```

University of Glasgow

Resource management with RAII

- This principle is applicable not just for memory management, but in general for *resource management*
  Resources might be: Files, Network and Database connections, ...
- For example we can encapsulate the opening and closing of a file using RAII:

```
struct file_handle {
  FILE * file;
  file_handle(const char * filename) {
    file = fopen(filename, "r");
    if (!file) { exit(EXIT_FAILURE); }
  }
  ~file_handle() { fclose(file); }
};
typedef struct file_handle file_handle;
int main() {
  file_handle f("filename"); // call to file_handle(filename) opens file
  // use f.file
} // call to ~file_handle() closes file
```

- RAII is used everywhere in C++ for resource management. We will see further examples later in the course

University of Glasgow

Modern Memory Management in C++

- C++ provides common data types build with RAII for easy and non-leaking memory management
- If we want to store multiple values of the same type on the heap we should use `std::vector`:

```
int main() {
  std::vector<int> v = {1, 2, 3, 4, 5, 6}; // heap memory is allocated for the numbers
  // use v
} // memory on the heap will be freed automatically
```

- If we want to store a single value on the heap we should use one of two "smart" pointers:
  - `std::unique_ptr` for unique ownership
  - `std::share_ptr` for shared ownership
- Unique ownership should be the default case where a single variable is *owning* the value on the heap
- Shared ownership should be used in situations where it is not possible to identify a single owner
  This is common in multi-threaded applications and we will see examples of this later in the course

University of Glasgow

Binary Tree with Smart Pointers (example of unique ownership)

```
#include <memory>
struct node {
  const void * value_ptr;
  // The parent owns both children uniquely
  std::unique_ptr<struct node> left_child;
  std::unique_ptr<struct node> right_child;

  // Each child knows their parent, but doesn't own it
  // We use plain pointers for non-owning relationships
  struct node* parent;

  node(const void * value_ptr_) {
    value_ptr = value_ptr_;
    left_child = NULL; right_child = NULL; parent = NULL }

  // The destructors of left/right_child are called automatically and free their heap
memory
  ~node() { }

  void add_left_child(const void* value_ptr) {
    // make_unique allocates memory for a node on the heap and calls the node-
constructor
    left_child = std::make_unique<node>(value_ptr); }

  void add_right_child(const void* value_ptr) {
    right_child = std::make_unique<node>(value_ptr); }
};
```

University of Glasgow

DAG with Smart Pointers (example of shared ownership)

```
#include <memory>
#include <vector>

struct node {
  const char * value;
  // every node can have multiple outgoing edges
  // a node "owns" the nodes it point to
  // this is a shared ownership as a single node can have multiple nodes pointing to it
  std::vector< std::shared_ptr<node> > edges_to;
  // constructor
  node(const char * v) { value = v; }
  // add a new connection to the DAG
  void add_edge_to(std::shared_ptr<node> n) { edges_to.push_back(n); }
}; // we do not need a custom destructor as the shared_ptr manages its memory automatically

// create a diamond-shaped DAG
std::shared_ptr<node> create_diamond() {
  std::shared_ptr<node> a = std::make_shared<node>("a");
  a->add_edge_to(std::make_shared<node>("b"));
  a->add_edge_to(std::make_shared<node>("c"));
  std::shared_ptr<node> d = std::make_shared<node>("d");
  a->edges_to[0]->add_edge_to(d);
  a->edges_to[1]->add_edge_to(d);
  return a;
}
```

University of Glasgow

Ownership transfer

- When modelling unique ownership we should use a `std::unique_ptr`
- For example we can model a tree where the parent is uniquely responsible for its children like this:

```
struct node {
  const char * value;
  std::vector< std::unique_ptr<node> >
children;
  node(const char * v) { value = v; }
};
```

- When we add a member function to add children we receive an error:

```
struct node { /* ... */ void add_child(std::unique_ptr<node> n) {
children.push_back(n); } };
```

```
memory:1805:31: error:
call to implicitly-deleted copy constructor of 'std::__1::unique_ptr<node,
std::__1::default_delete<node> >'
```

- What are we trying to achieve here? The ownership should be unique, but if we make a copy of a unique pointer then clearly two variables (the original and the copy) hold ownership. The compiler is preventing that this happens!

Instead we have to explicitly *transfer the ownership*!

University of Glasgow

Ownership transfer with `std::move`

- To transfer the ownership of a unique resource such as `std::unique_ptr` we use `std::move`:

```
void add_child(std::unique_ptr<node> n) { children.push_back(std::move(n)); }
```

- The ownership transfer signals that the old variable is no longer responsible to manage the resource:

```
void main() {
  std::unique_ptr<node> a_ptr = std::make_unique<node>("a");
  {
    std::unique_ptr<node> b_ptr = std::make_unique<node>("b");
    a_ptr->add_child(std::move(b_ptr)); // ownership is transferred here
  } // lifetime of b_ptr ends => nothing happens (b_ptr doesn't own anything
anymore)
} // lifetime of a_ptr ends => memory allocated for a and b will be freed
```

- It is forbidden to access the value of a variable for which ownership has been transferred

```
  std::unique_ptr<node> b_ptr = std::make_unique<node>("b");
  a_ptr->add_child(std::move(b_ptr));
  b_ptr->value; // BAD IDEA
```

University of Glasgow

Next week

- We are going to discuss a bit more about the ownership model and **transfer of ownership**
- We are going to look at **debugging** and **development tools**
- We will learn how debuggers help us to detect **bugs**
- We will learn how **static and dynamic analysis tools** help us to address problems like memory leaks

University of Glasgow

Task

- **There are two Lecture 4 Example folders on moodle, containing both C (.c) and C++ (.cpp) files**
- **T1: try to execute the examples of both c and c++**
- **T2: notice the differences between binary_tree.cpp and binary_tree.cpp**
- **T3: bring your notes to the Zoom meeting**

University of Glasgow

Systems Programming
Debugging and Development Tools

Anna Lito Michala

Email AnnaLito.Michala@Glasgow.ac.uk

University of Glasgow

Recap from last week: Ownership

- To organise resource and memory management we adopt the concept of *ownership*
- In C++ we tie the management of a resource to the lifetime of a variable on the stack. The allocation of the resource (e.g. the malloc call) is done when we create the variable. The deallocation of the resource (e.g. the free call) is done when the variable is destroyed.

```
int main() {
std::vector<int> v = {1, 2, 3, 4, 5, 6}; // heap memory is allocated for the numbers
// use v
} // memory on the heap will be freed automatically
```

- In C++ this idiom is implemented by struct data types with two *member functions* (called methods in Java):
  - The *constructor* which is called when creating values
  - The *destructor* which is called when a variable of this type reaches its lifetime's end
- C++ provides common data types for easy and non-leaking memory management:
  - std::vector for ownership of multiple values
  - std::unique_ptr for unique ownership of a single value
  - std::share_ptr for shared ownership of a single value

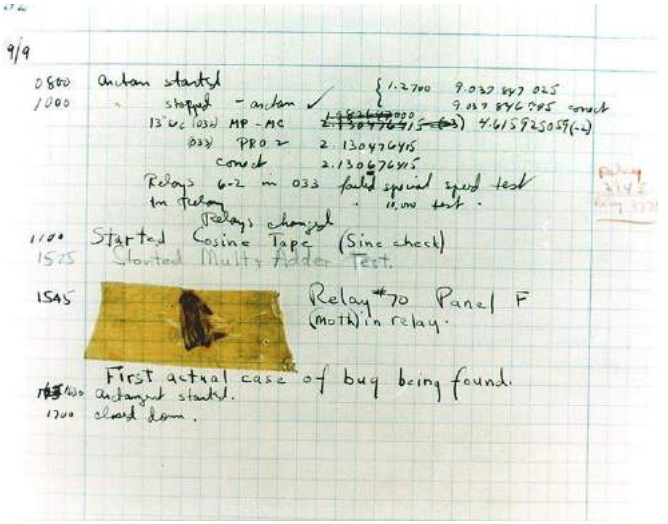
Bugs and finding them

- Software development is not free of errors
- We call these software errors *bugs*
- We will investigate a number of tools to help us identify and understand bugs:

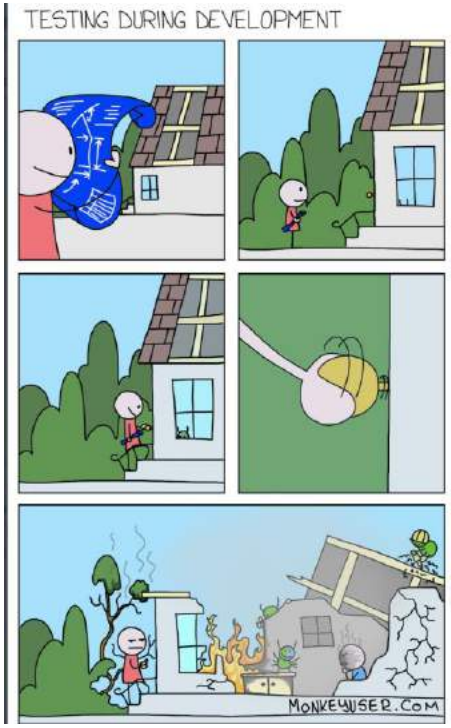
– **Debuggers**, run a program in a controlled environment where we can investigate its execution

– **Static analysis** tools, reason about a program's behaviour without running it

– **Dynamic analysis** tools, add instructions to a program to detect bugs at runtime



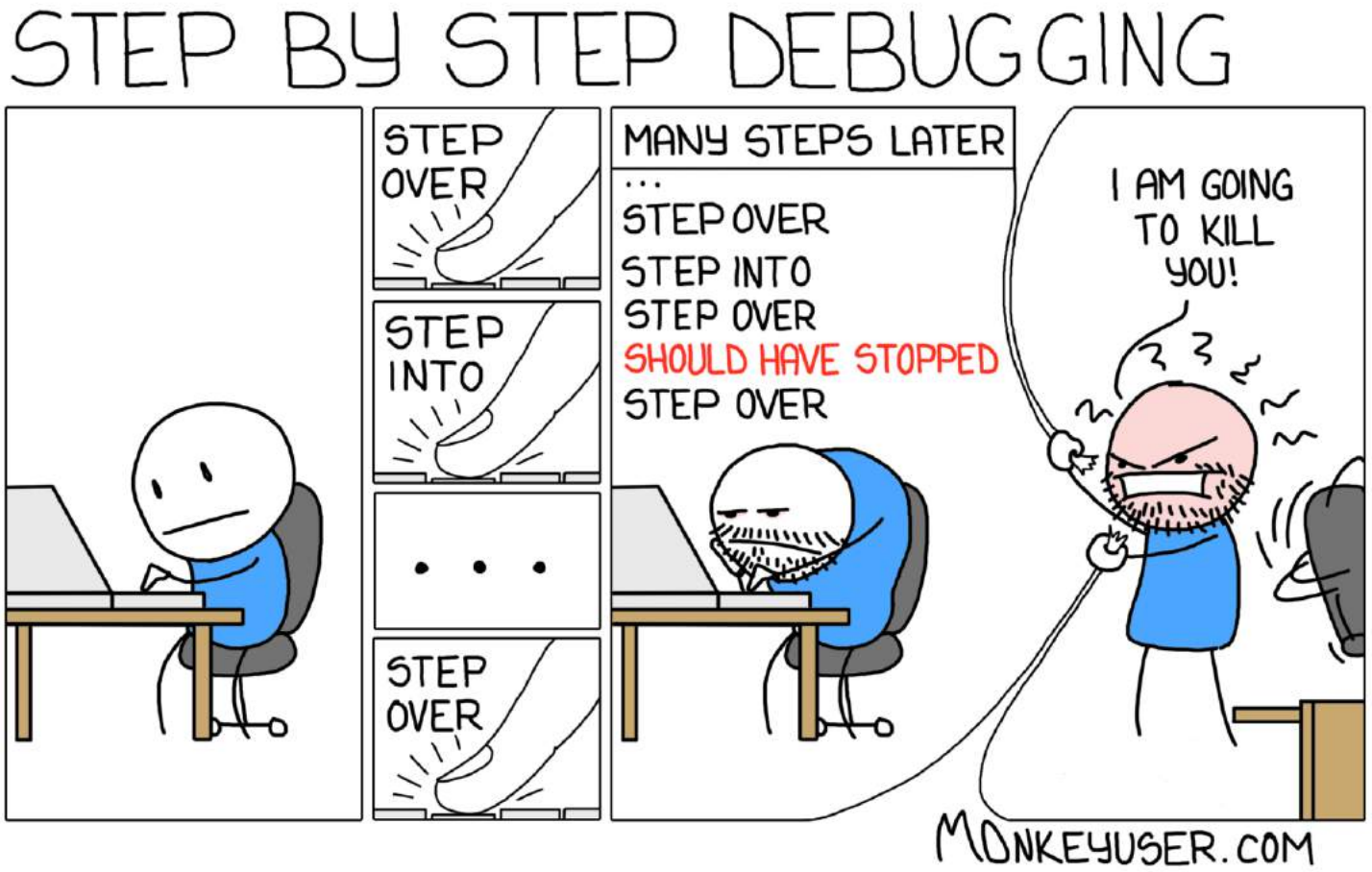
Logbook of Grace Hoppers team in 1



Monkeyuser.com

University of Glasgow

Debugger



University of Glasgow

Debugger

- The two popular debuggers for C/C++ programs `GDB` and `LLDB` are used almost exactly the same

1. We need to compile the program with the `-g` flag which adds debug information into the binary
2. Instead of executing the program normally we start debugger and load the program:

```
$ ./program arg1 arg2 ; "Run program normally"
$ gdb --arg ./program arg1 arg2 ; "Load program in GDB debugger"
$ lldb -- ./program arg1 arg2 ; "Load program in LLDB debugger"
```

3. Inside the debugger we start the execution with the `run` command:

```
(lldb) run
```

- A good list of commands available in the `GDB` and `LLDB` debuggers is available here: https://lldb.llvm.org/lldb-gdb.html
- To exit run the `quit` command inside the debugger

University of Glasgow

Segmentation fault

- This runtime error message is one of the most common in systems programming:

```
[1] 83437 segmentation fault ./program
```

- A *segmentation fault* is raised by the hardware notifying the operating system that your program has attempted to access a restricted area of memory
- The operating system will then immediately terminate your program
  In Operating Systems you will learn how it is possible to catch the segmentation fault signal and handle it.
- The most common examples of programs with `segfault` are:

– **Dereferencing a NULL pointer**

– **Writing to read-only memory**

– **A buffer overflow**, i.e. accessing memory outside of an allocated buffer

– **A stack overflow**, often triggered by a recursion without a base case

University of Glasgow

Where did my segmentation fault come from?

- To find the line in a program that triggered the segmentation fault we load it in the debugger and run it:

```
$ lldb -- ./program 12345
(lldb) run
Process 85058 launched: './program' (x86_64)
2018-10-21 20:56:00.106714+0100 program[85058:12827554] detected buffer
overflow
Process 85058 stopped
```

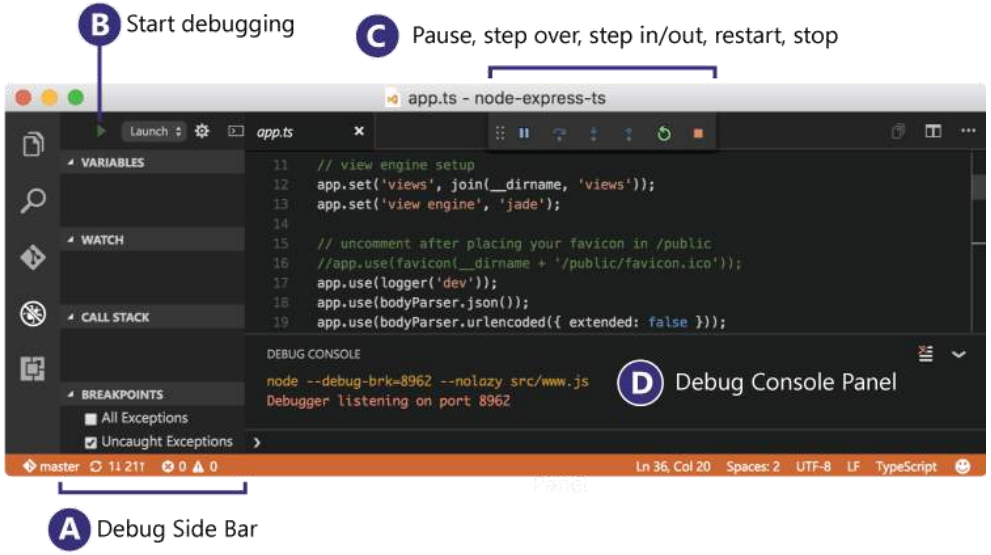
- The debugger has now stopped the execution.
  Using the `bt` (*backtrace*) command we investigate the calls leading to the segfault:

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGABRT
* frame #0: 0x00007fff76d1ab86 libsystem_kernel.dylib`__pthread_kill + 10
...
frame #6: 0x00007fff76ca8e84 libsystem_c.dylib`__strcpy_chk + 8
frame #7: 0x0000000100000eaf program overflow(argc=2, argv=0x00007ffeefbff300) at
program.c:35
frame #8: 0x0000000100000efb program`main(argc=2, argv=0x00007ffeefbff3c8) at program.c:42
frame #9: 0x00007fff76bdc085 libdyld.dylib`start + 1
frame #10: 0x00007fff76bdc085 libdyld.dylib`start + 1
```

- Here *frame 7* shows us the file (`program.c`) and line (35) which triggered the segfault in our code

Breakpoints and GUI for debugging

- *Breakpoints* are points at which the execution is stopped and we can investigate the state of the execution
- Setting breakpoints in the command line debugger is tedious (but totally possible)
- The `Visual Studio Code editor` (but also others, like Atom) provide a GUI for the `GDB` or `LLDB` debugger



Detailed instructions for the configuration can be found here:

https://code.visualstudio.com/docs/editor/debugging

University of Glasgow

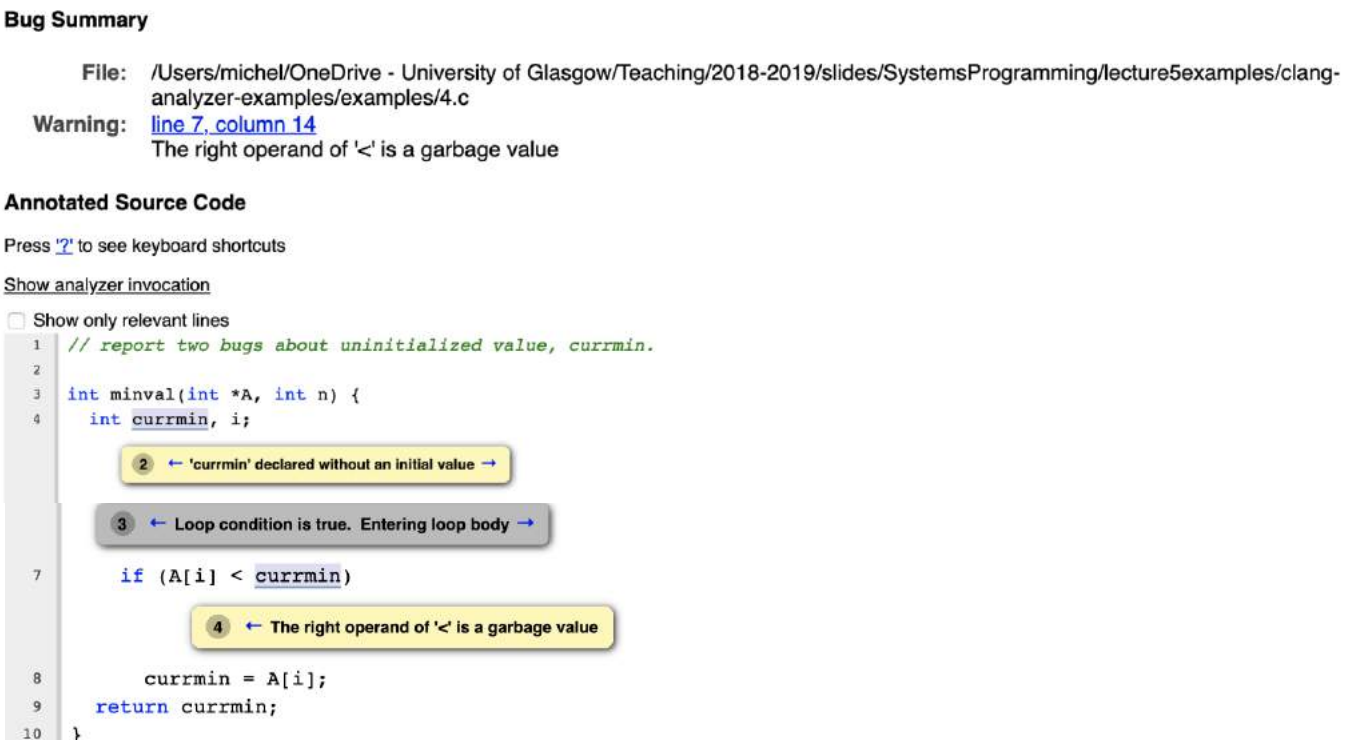
Static Analysis

- A static analysis reasons about the code without executing it
- The compiler performs some static analysis every time you compile your code, e.g. type checking
- It is good practice to enable all warnings (`-Wall`) and make warnings errors (`-Werror`)
  This way the compiler can be most helpful in detecting bugs before your program is executed
- Some static analysis is too expensive to perform in every build
  Other static analysis enforces a particular coding guideline
- These type of static analysis is available with special flags or in separate tools
- We are going to use the tools provided by the clang compiler
- We invoke the static analyzer using a flag and specifying the output format of the report:

```
clang --analyze --analyzer-output html program.c
```

Clang static analyzer report

- The tool generates a report explaining the potential bug



- Here has the tool reported a "garbage value" which in this case leads to undefined outcome for a if-branch

University of Glasgow

clang-tidy

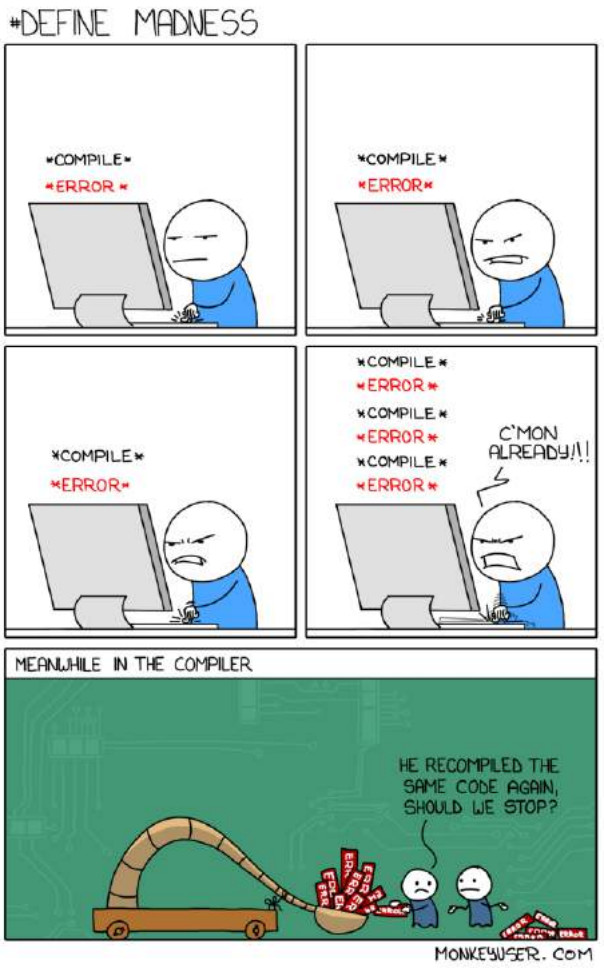
- `clang-tidy` is a *linter* (the name comes from the first UNIX tool to perform static analysis on C). It is invoked like a compiler accepting the same flags as `clang` specified after two dashes: `--`
- A series of *checks* can be en- and disabled. Here we enable the checks for *readability*:

```
$ clang-tidy -checks="readability-*" 6.c -- -Wall -Werror
/Users/lito/Desktop/examples/6.c:2:16: warning: pointer parameter 'p' can be pointer to const
[readability-non-const-parameter]
void test(int *p) {
           ~~~ ^
          const
/Users/lito/Desktop/examples/6.c:4:9: warning: statement should be inside braces
[readability-braces-around-statements]
if (p)
      ^
       {
```

- We are suggested to put braces around the branch of an if-statement and to use `const`
- `clang-tidy` is a flexible tool that allows to enforce coding guidelines and to modernize source code. It is possible to extend `clang-tidy` by writing your own checks
- Detailed information for `clang-tidy` are available here: http://clang.llvm.org/extra/clang-tidy/

University of Glasgow

Use the analyzer results to improve your code



University of Glasgow

Dynamic Analysis Tools

- There exists a family of bug detection tools which use dynamic analysis
- These tools need the program to run and can only detect bugs which are encountered during the execution of a particular test input
- The `clang` project calls these tools *sanitizers*. The most important are:
  - `AddressSanitizer` - a memory error detector
  - `MemorySanitizer` - a detector of uninitialized reads
  - `LeakSanitizer` - a memory leak detector
  - `UndefinedBehaviorSanitizer` - a detector of undefined behaviour
- We will later in the course use:
  - `ThreadSanitizer` - a data race detector

University of Glasgow

Address Sanitizer

- Address Sanitizer is a memory error detector for detecting:
  - Out-of-bounds accesses
  - Use-after-free
  - Double free
- By enabling this tool clang will insert instructions in the program to monitor every single memory access
- This slows down the execution by about 2x. The similar tool valgrind has often a slowdown of 20-100x!
- These flags enable address sanitizer:

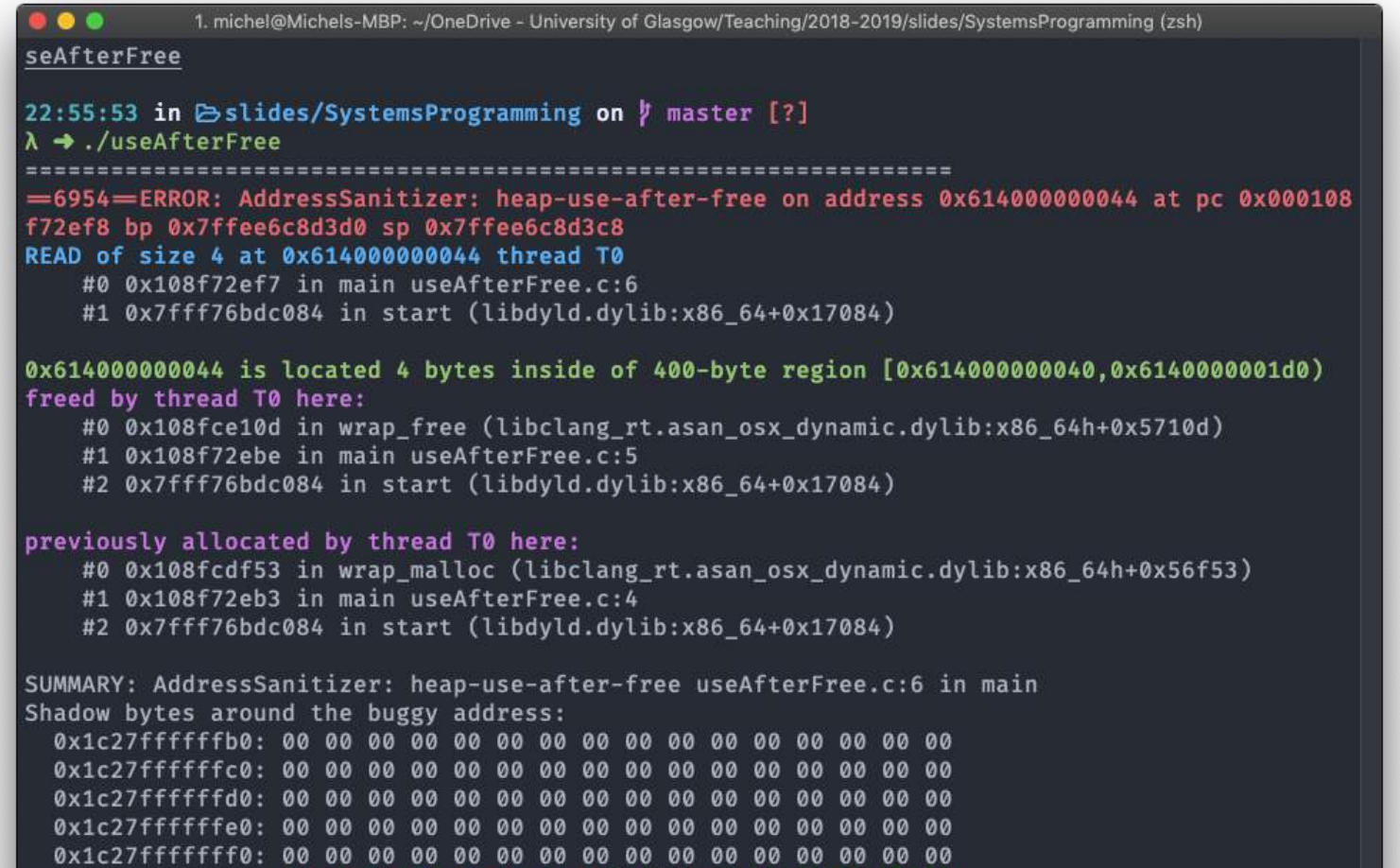
```
clang -fsanitize=address -fno-omit-frame-pointer -O1 -g -Wall -Werror program.c -o program
```

- The `-fno-omit-frame-pointer` flag produces a readable call stack and `-O1` enables basic optimizations.
- The compiler will produce a binary as usual which we execute: `./program`
- Address Sanitizer has found hundreds of bugs in large scale software (e.g. Chromium and Firefox)

University of Glasgow

Address Sanitizer output

- The output reports a heap-use-after-free on address 0x614000000044 and provides information where the memory was freed (line 5) and allocated (line 4)



University of Glasgow

Memory Sanitizer

- Memory sanitizer detects uninitialized reads to memory:

```
% cat umr.c
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char** argv) {
int* a = malloc(sizeof(int)*10);
a[5] = 0;
if (a[argc])
printf("xx\n");
return 0;
}
% clang -fsanitize=memory -fno-omit-frame-pointer -g -O2 umr.cc
% ./a.out
WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x7f45944b418a in main umr.c:6
#1 0x7f45938b676c in __libc_start_main libc-start.c:226
```

- This tool is under active development and currently only available for linux

University of Glasgow

Leak Sanitizer

- Leak sanitizer detects memory leaks, that is memory which hasn't been freed at the end of the program:

```
$ cat memory-leak.c
#include <stdlib.h>
void *p;
int main() {
p = malloc(7);
p = 0; // The memory is leaked here.
return 0;
}
% clang -fsanitize=address -g memory-leak.c
% ./a.out
==23646==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 7 byte(s) in 1 object(s) allocated from:
#0 0x4af01b in __interceptor_malloc /projects/compiler-
rt/lib/asan/asan_malloc_linux.cc:52:3
#1 0x4da26a in main memory-leak.c:4:7
#2 0x7f076fd9cec4 in __libc_start_main libc-start.c:287
SUMMARY: AddressSanitizer: 7 byte(s) leaked in 1 allocation(s).
```

- This tool is also under active development and marked as experimental

University of Glasgow

Undefined Behavior

- What is undefined behaviour?
- For certain operations the C standard demands no particular behaviour.
  These are typically cases which are considered bugs, e.g. dereferencing a `null` pointer.
- It is expensive to check every time a pointer is dereferenced if the operation is valid.
  Because the compiler does not have to ensure a particular behaviour for `null` pointers, it can *assume* that the programmer has ensured that the pointer is not `null`.
- Undefined behaviour is therefore crucial for fast code, but makes detection of bugs much harder, as it is not guaranteed that a bug will result in a program crash
- A good introduction to undefined behaviour is this series of blog posts:
  What Every C Programmer Should Know About Undefined Behavior

University of Glasgow

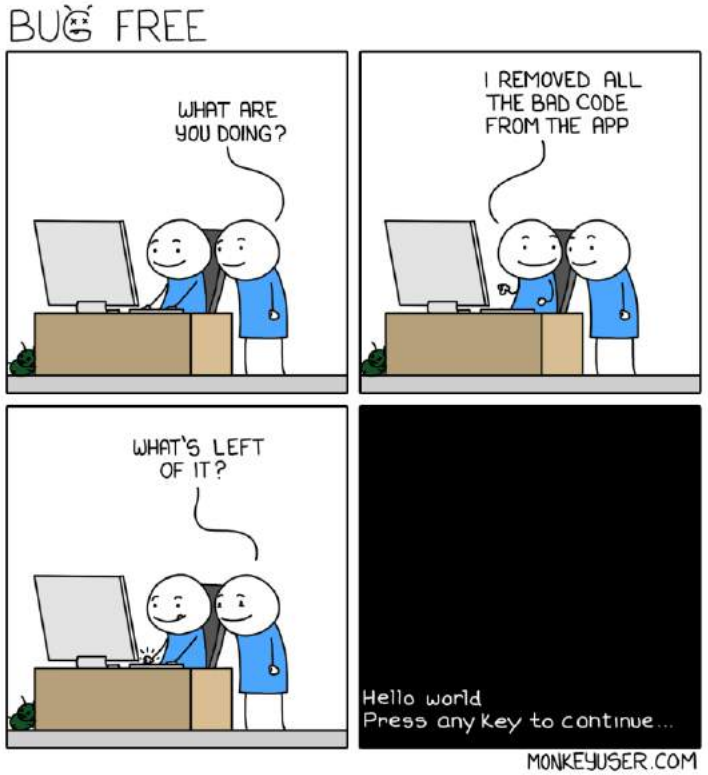
Undefined Behavior Sanitizer

- Undefined behavior sanitizer detects various types of undefined behaviour
- Here an integer overflow is detected:

```
% cat intOverflow.c
int main(int argc, char **argv) {
int k = 0x7fffffff; // this is the largest possible signed int value ...
k += argc; // ... this will produce an integer overflow
return 0;
}
% clang -fsanitize=undefined -Wall -Werror intOverflow.c -o intOverflow
% ./intOverflow
intOverflow.c:3:5: runtime error: signed integer overflow: 2147483647 + 1
cannot
be represented in type 'int'
```

Next week

- **There is no lecture next week!**
- You are invited to work on your coursework in the lab
- In two weeks, we will start looking at threads and how to write multi-threaded applications



University of Glasgow

Systems Programming (Part 2)

Concurrent Systems Programming

Phil Trinder

University of Glasgow

Recap of lectures so far

In the introduction to systems programming, we have learned that:

- **data types** give bit representations in memory a meaning
- **structs** allow us to implement custom data structures (like linked lists or trees)
- every variable is stored at a fixed **memory location**
- a **pointer** is a variable storing the address of a memory location
- in C/C++ memory on the **stack** is automatically managed but memory on the **heap** must be managed manually
- to organise resource and memory management we should think about **ownership**
- in C++ ownership is implemented by tying the resource management to the **lifetime** of a stack variable
- **smart pointers** and **containers** make it easier manage memory, and to avoid leaks
- **debuggers**, **static** and **dynamic analysis tools** help to detect and fix bugs
- In the second part of the course we are looking into **concurrent systems programming**

University of Glasgow

What is concurrency?

- The ability of different program parts (typically functions) to be executed simultaneously.
- There are many different methods to enable concurrent programs. We distinguish two classes:
  - *Shared memory* locations are read and modified to communicate between concurrent components. Requires *synchronisation* to ensure that communication happens safely.
    We will look at explicit programming with *threads* and use shared memory communication.
  - *Message passing* tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust form of concurrent programming. Examples of message passing systems are the *actor model* implemented in *Erlang* or *CSP*-style communication in Go.
    Covered in the *Distributed and Parallel Technologies H/M* course

Concurrency vs. Parallelism

We make a clean distinction between Concurrency and Parallelism. Concurrency is a _programming paradigm_, wherein threads are used typically for dealing with multiple asynchronous events fromm the environment, or for structuring your program as a collection of interacting agents. Parallelism, on the other hand, is just about making your programs go faster. You shouldn't need threads to do parallelism, because there are no asynchronous stimuli to respond to. It just so happens that it's possible to run a concurrent program in parallel on a multiprocessor, but that's just a bonus. I guess the main point I'm making is that to make your program go faster, you shouldn't have to make it concurrent. Concurrent programs are hard to get right, parallel programs needn't be.

Simon Marlow on the Haskell-cafe mailing list

But when people hear the word concurrency they often think of parallelism, a related but quite distinct concept. In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations. Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.

Andrew Gerrand on the Go blog

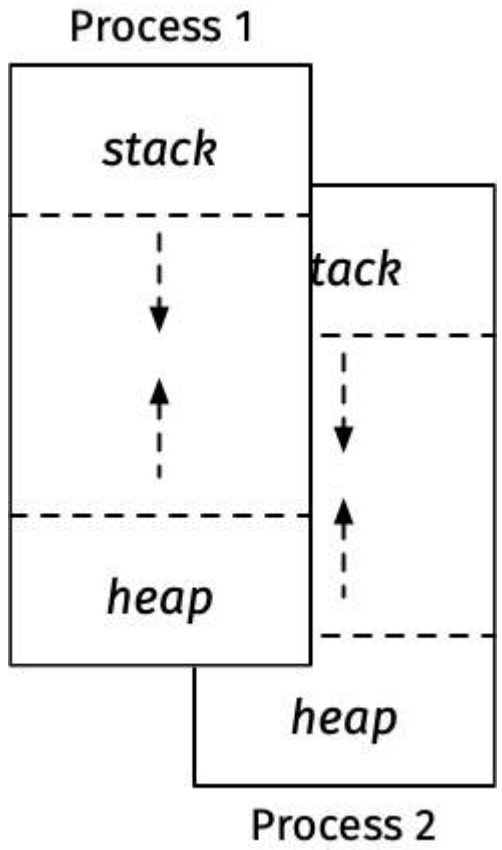
Processes vs. Threads

Processes

- Every C program we have written is executed as a process
- Multiple processes can be executed simultaneously
- Each process has its own memory address space

Threads

- A thread of execution is an independent sequence of program instructions
- Multiple threads can be executed simultaneously
- A process can have multiple threads sharing the same address space of the process
- We will use threads to implement concurrent programs



University of Glasgow

Thread implementations

- There are thread implementations in almost all programming languages
- We will use the C pthread library and in the next lecture we will use C++ threads
- Many threading implementations are conceptually very similar, and follow a lifecycle:
  - A thread is *created*
    - starts executing a specified function
    - with some arguments
    - Is given an identifier
  - Wait for another thread to terminate
  - Interrupt/kill another thread
  - Terminates either by explicitly calling *exit* or when its function terminates
- Communication between threads is by modifying the state of *shared variables*

University of Glasgow

POSIX Threads

- POSIX Threads (short pthreads) is the most commonly used threading implementation for C
- To use it we need to `#include <pthread.h>`

  and specify a compiler flag `-lpthread`

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

void * PrintHelloWorld(void*) {
printf("Hello World from a thread!\n");
return NULL;
}
```

```
int main() {
pthread_t thread;
int error = pthread_create(&thread, NULL,
PrintHelloWorld, NULL) ;
assert(error == 0);
printf("Created thread\n");
error = pthread_join(thread, NULL);
assert(error == 0);
}

clang -Wall -Werror program.c -lpthread -o program
```

University of Glasgow

Creating a thread

Threads are created with the pthread_create function

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine)(void*),
        void *arg);
```

It takes four arguments:

1. A thread identifier, i.e. a pointer to a memory location of type pthread_t.
2. Thread attributes which set properties such as scheduling policies or stack size.
   Passing NULL results in default attributes.
3. A function pointer to the start_routine.
   This function takes a single argument of type void* and returns a value of type void*.
4. The argument that is passed to start_routine.

It returns 0 if the thread is created successfully or a non-zero error code.

Passing pointers to and from start_routine allows the passing of arbitrary data.

It also requires care to ensure that the memory locations pointed to have appropriate lifetimes.

Waiting for another thread to Terminate

To wait for another thread to terminate we use pthread_join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

It takes two arguments:

1. A thread identifier.
2. A pointer to a memory location of type void*.

   The return value of the start_routine passed to pthread_create will be copied to this location.

It returns 0 on success and a non-zero error code otherwise.

Waiting for another thread to Terminate

To wait for another thread to terminate we use pthread_join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

It takes two arguments:

1. A thread identifier.
2. A pointer to a memory location of type void*.

   The return value of the start_routine passed to pthread_create will be copied to this location.

It returns 0 on success and a non-zero error code otherwise.

Example of returning a single int value from a thread:

```
int* return_value;
int error = pthread_join(thread, (void**)&return_value);
assert(error == 0);
if (return_value) { printf("return_value: \n", *return_value); }
// maybe: free(return_value);
```

University of Glasgow

To Do Now

Completing these activities will reinforce your understanding of the material, and reinforce your understanding of the material

In the “Examples from Lecture 6” folder on Moodle

- Review the code for pthread_hello_world.c
- Compile it and and run it
- Add another print statement “Last thread about to die” just before the main thread terminates

Introducing Mutual Exclusion

- The Mutual Exclusion problem was first identified by *Edsger Dijkstra* in 1965
- This problem (and its solution) is the beginning of the computer science of concurrency
- What is the problem that mutual exclusion is solving?
- Consider two threads T0 and T1 attempting to increment a `count` variable with a C compiler that implements `count++` as

```
register1 = count
register1 = register1 + 1
count = register1
```

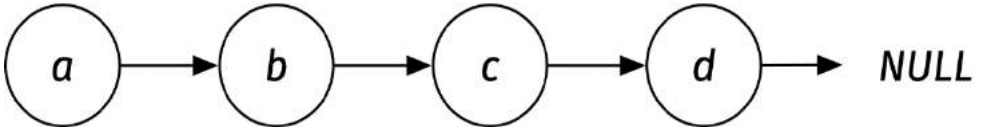
- Consider this execution interleaving with "count = 5" initially:

```
T0: register1 = count          {T0 register1 = 5}
T0: register1 = register1 + 1  {T0 register1 = 6}
T1: register1 = count          {T1 register1 = 5}
T1: register1 = register1 + 1  {T1 register1 = 6}
T1: count = register1          {count = 6 }
T0: count = register1          {count = 6}
```

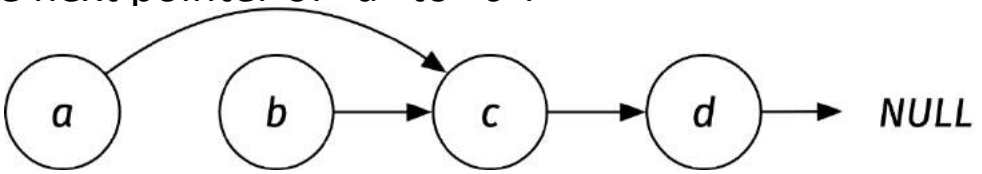
Q: Should the value of `count` be 6? What has gone wrong? How can we fix it?

Mutual Exclusion Example 2

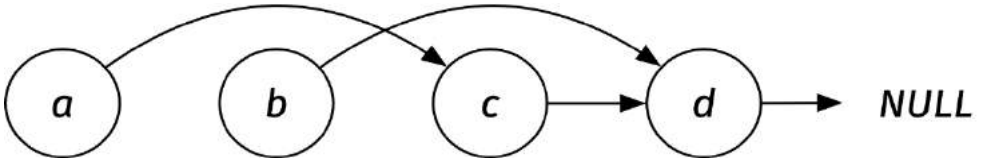
- Consider a singly linked list:



- Two threads simultaneously start removing the elements from the list.
- The first thread removes “b” by moving the next pointer of “a” to “c”:



- The second thread removes “c” by moving the next pointer of “b” to “d”:



- The resulting list is inconsistent, i.e. it still contains node “c” (the item deleted by the second thread)!
- We can avoid inconsistency by ensuring that simultaneous updates to the same part of the list cannot occur

University of Glasgow

Critical Regions & Race Conditions

- A critical region is a part of the code that updates some shared variable(s) (like `count`), or a shared data structure (like the list)
- Critical regions must be protected so that concurrent threads don't interfere with each other. Mutual exclusion means that only one thread ever executes a critical section
- A race condition occurs when the result of program execution depends on the order in which threads are executed, e.g. a different interleaving of T0 and T1 with "count = 5" would produce:

```
T0: register1 = count    {T0 register1 = 5}
T0: register1 = register1 + 1    {T0 register1 = 6}
T0: count = register1    {count = 6}
T1: register1 = count    {T1 register1 = 6}
T1: register1 = register1 + 1    {T1 register1 = 7}
T1: count = register1    {count = 7 }
```

Q: Demonstrate a race condition for the list deletion

University of Glasgow

Locks Protect Critical Regions

- Associate a lock (aka a mutex) with each critical section
- Before executing a critical section a thread must acquire the lock
- On leaving the critical section the thread must release the lock
- The semantics of acquire are:

  if another thread owns the lock, the requesting thread is blocked until the owning thread releases the lock

  If the lock is not owned, the requesting thread is granted ownership
- On release of a lock, *one of the threads* (if any) blocked waiting for the lock will be granted ownership and continue execution

University of Glasgow

Simple Lock Example

- If both threads T0 and T1 execute:

```
acquire(theLock);
count++;
release(theLock);
```

the threads will execute without interfering

- That is either T0 will complete before T1, or T1 before T0

University of Glasgow

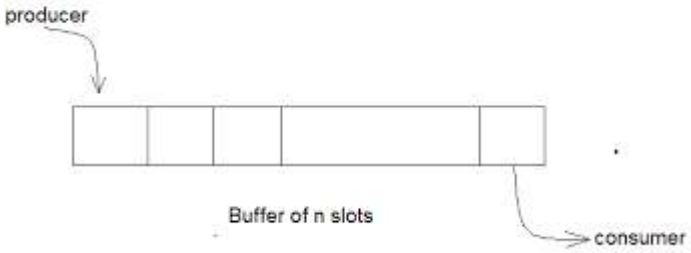
Bounded Buffer

- Problem first proposed by Dijkstra in 1972
- A bounded buffer is a buffer with a fixed size
- *Producers* insert elements into the buffer
- *Consumers* remove elements from the buffer
- We have to ensure that:

  producers wait when the buffer is full.

  consumers wait when the buffer is empty.

  threads resume work when space or data is available again.



Unsafe Bounded Buffer

```
int count = 0;
int in = 0;
int out = 0;
Type buffer[N];
```

```
//PRODUCER
for (;;) {
 // produce an item and put in nextProduced
    while (count == N)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % N;
    count++;
}
```

```
//CONSUMER
for (;;) {
    while (count == 0)
        ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % N;
    count--;
    //  consume the item in nextConsumed
}
```

Q: Show two interactions of producer and consumer that can lead to inconsistency

University of Glasgow

Locked Bounded Buffer

```
Lock theLock;
int count = 0;
int in = 0;
int out = 0;
Type buffer[N];
```

```
//PRODUCER
for (;;) {
    // produce an item and put in nextProduced
    acquire(theLock);
    while (count == N)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % N;
    count++;
    release(theLock);
}
```

```
//CONSUMER
for (;;) {
    acquire(theLock);
    while (count == 0)
        ; // do nothing
    nextConsumed =  buffer[out];
    out = (out + 1) % N;
    count--;
    release(theLock);
    //  consume the item in nextConsumed
}
```

In this version the buffer will not become inconsistent. Are we done?

University of Glasgow

Locks can cause Deadlock

- Consider the situation where `count == 0` and the Consumer executes

```
while (count == 0)
    ;
```

- While executing this loop, the Consumer owns theLock, so the Producer cannot acquire it to place a new item in the buffer and increment `count`
- As a result, we have deadlock: neither thread can make progress
- Q: Show a different case where the Locked Bounded Buffer can deadlock

University of Glasgow

Busy Waiting Deadlock Solution

- We can avoid deadlock if when a thread must wait for some condition it releases the locks it owns
- So in our example the consumer repeatedly:

  Acquire Lock

  If (count == 0) Release Lock
- This is called Busy Waiting (or sometimes polling)
- Wastes CPU cycles and energy!

Tea maker with busy waiting

```
pthread_mutex_t m;
bool teaIsReady = false;

void *me(void* arg) { (void)arg;
  printf("Me: Waiting for my tea ...\n");
  pthread_mutex_lock(&m);
  while (!teaIsReady) {
    pthread_mutex_unlock(&m);
    printf("Me: (Unamused) // do nothing\n");
    pthread_mutex_lock(&m); }
  pthread_mutex_unlock(&m);
  printf("Me: (Happy) ... finished waiting.\n");
  return NULL; }
```

```
void *teaRobot(void* arg) { (void) arg;
  printf(" Tea Robot: Making tea ...\n");
  usleep(randInt());
  pthread_mutex_lock(&m);
  teaIsReady = true;
  pthread_mutex_unlock(&m);
  printf(" Tea Robot: Done!\n");
  return NULL; }
```

```
int main() {
  pthread_t t1; pthread_t t2; int err;
  err = pthread_mutex_init(&m, NULL); assert(err == 0);
  err = pthread_create(&t1, NULL, me, NULL); assert(err == 0);
  err = pthread_create(&t2, NULL, teaRobot, NULL); assert(err == 0);
  err = pthread_join(t1, NULL); assert(err == 0);
  err = pthread_join(t2, NULL); assert(err == 0);
  pthread_mutex_destroy(&m); }
```

University of Glasgow

Condition Variable Deadlock Solution

- Busy waiting wastes CPU cycles and energy
- It would be better to block the thread seeking to acquire a lock, and wake it when it has a chance to proceed
- Achieved using a condition variable `cv`:

`pthread_cond_wait(&cv, &m)`

must be called with a locked mutex `m`

It releases the mutex *and* blocks the calling thread on cv

`pthread_cond_signal(&cv)`

assigns mutex ownership to *one* of the threads blocked on `cv`, and wakes it

University of Glasgow

Condition Variable Deadlock Solution

- The condition may have changed before the wakened thread is scheduled, so it's important to check the condition again:

```
pthread_mutex_lock(&m);
while (!cond) {
  pthread_cond_wait(&cv, &m);}
```

- Condition variables are analogous to hardware signals/interrupts

University of Glasgow

Tea maker with condition variables

```
pthread_mutex_t m; pthread_cond_t cv;
bool teaIsReady = false;

void *me(void* arg) { (void)arg;
  pthread_mutex_lock(&m);
  while (!teaIsReady) {
    printf("Me: Waiting for my tea ...\n");
    pthread_cond_wait(&cv, &m); }
  printf("Me: (Happy) ... finished waiting.\n");
  printf("Me: Is the tea really finished? %s\n",
  teaIsReady ? "Yes" : "No");
  pthread_mutex_unlock(&m);
  return NULL; }

void *teaRobot(void* arg) { (void) arg;
  printf(" Tea Robot: Making tea ...\n");
  usleep(randInt());
  pthread_mutex_lock(&m);
  teaIsReady = true;
  pthread_mutex_unlock(&m);
  pthread_cond_signal(&cv);
  printf(" Tea Robot: Done!\n");
  return NULL; }

int main() {
  pthread_t t1; pthread_t t2; int err;
  err = pthread_mutex_init(&m, NULL); assert(err == 0);
  err = pthread_cond_init(&cv, NULL) ; assert(err == 0)
  err = pthread_create(&t1, NULL, me, NULL); assert(err == 0);
  err = pthread_create(&t2, NULL, teaRobot, NULL); assert(err == 0);
  err = pthread_join(t1, NULL); assert(err == 0); err = pthread_join(t2, NULL); assert(err == 0);
  pthread_cond_destroy(&cv) ; pthread_mutex_destroy(&m); }
```

University of Glasgow

To Do Now

Attempt the questions on the preceding slides

In the “Examples from Lecture 6” folder on Moodle

- Review the code for the tea maker examples tea1.c, tea2.c and tea3.c
- Compile and run the examples a couple of times
- Compare the results, and reflect on them

University of Glasgow

Bounded Buffer With Condition Variables

- We need two condition variables:
  - Producer must wait until there is space in the buffer: `add`
  - Consumer must wait until there are item(s) in the buffer: `remove`
- This bounded buffer is a monitor – a class that provides thread safe access to a shared resource (the buffer).
- In a monitor all (public) functions, like `addItem, removeItem` provide mutual exclusion
- A Java class where all public methods are `synchronised` is a monitor

University of Glasgow

Bounded Buffer Monitor Main

```
struct BoundedBuffer {
  int start; int end; int size; int* buffer;
  pthread_mutex_t m;
  pthread_cond_t add;
  pthread_cond_t remove;
};

typedef struct BoundedBuffer BoundedBuffer;

BoundedBuffer * createBoundedBuffer(int size) { ... }
void destroyBoundedBuffer(BoundedBuffer * bb) { ... }
void addItem(BoundedBuffer * bb, int item) { ... }
int removeItem(BoundedBuffer * bb) { ... }
void * producer(void * arg) { ... }
void * consumer(void * arg) { ... }
```

```
int main() {
pthread_t t1; pthread_t t2; int err;
BoundedBuffer * bb = createBoundedBuffer(4);
err = pthread_create(&t1, NULL, consumer, bb); assert(err == 0);
err = pthread_create(&t2, NULL, producer, bb); assert(err == 0);
err = pthread_join(t1, NULL); assert(err == 0);
err = pthread_join(t2, NULL); assert(err == 0);
}
```

University of Glasgow

Bounded Buffer Monitor Producer

```
void * producer(void * arg) {
  BoundedBuffer * bb = (BoundedBuffer*)arg;
  for (int i = 0; i < 10; i++) {
    int item = randInt();
    printf("produced item %d\n", item);
    addItem(bb, item);
    usleep(randInt());
  }
  return NULL;
}
```

University of Glasgow

Bounded Buffer Monitor Consumer

```
void * consumer(void * arg) {
  BoundedBuffer * bb = (BoundedBuffer*)arg;
  for (int i = 0; i < 10; i++) {
    int item = removeItem(bb);
    printf(" consumed item %d\n", item);
    usleep(randInt());
  }
  return NULL;
}
```

University of Glasgow

Bounded Buffer Monitor AddItem

```
void addItem(BoundedBuffer * bb, int item) {
  If (!bb) return;
  pthread_mutex_lock(&bb->m);
  while (bb->start == bb->end) { // buffer is full
    printf("== Buffer is full ==\n");
    pthread_cond_wait(&bb->add, &bb->m);
  }
  // buffer is no longer full
  bb->buffer[bb->start] = item;
  bb->start = (bb->start + 1) % bb->size; // move start one forward
  pthread_mutex_unlock(&bb->m);
  pthread_cond_signal(&bb->remove);
}
```

University of Glasgow

Bounded Buffer Monitor RemoveItem

```
int removeItem(BoundedBuffer * bb) {
  if (!bb) assert(0);
  pthread_mutex_lock(&bb->m);
  while ( ((bb->end + 1) % bb->size) == bb->start ) { // buffer is  empty
    printf("== Buffer is empty ==\n");
    pthread_cond_wait(&bb->remove, &bb->m);
  }
  // buffer is no longer empty
  bb->end = (bb->end + 1) % bb->size; // move end one forward
  int item = bb->buffer[bb->end];
  pthread_mutex_unlock(&bb->m);
  pthread_cond_signal(&bb->add);
  return item;
}
```

University of Glasgow

Bounded Buffer Monitor Reflection

- The buffer is implemented as a ring buffer
- Two condition variables (add and remove) signal the two different conditions (buffer is full or empty)

  Q: Can you implement the buffer with a single condition variable?

  Q: Do you recommend implementing the buffer with a single condition variable?
- Before inserting or removing an element the size of the buffer is checked and the threads wait if there is no space (producer) or no data (consumer)
- After inserting or removing an element the thread signals that data or space is now available
- The producer and consumer functions to not need to use locking themselves they use a monitor

University of Glasgow

To Do Now

Attempt the questions on the preceding slides

In the “Examples from Lecture 6” folder on Moodle

- Review the code for the bounded buffer:

  bounded_buffer_unsafe.c

  bounded_buffer_safe.c

- Compile and run the examples a couple of times
- Compare the results, and reflect on them

Concurrent Programming is Hard

A concurrent poses all of the challenges of sequential **Computation**: i.e. **what** to compute. A correct and efficient Algorithm, using appropriate data structures must be constructed.

A concurrent program must **also** specify a correct and effective strategy for **Coordination**: i.e. **how** threads should cooperate.

University of Glasgow

Some Important Coordination Aspects

Partitioning: determining what parts of the computation should be separately evaluated, e.g. a thread to serve each request, to render each frame of a film.

Placement: determining where threads should be executed, e.g. allocate thread to least busy core.

Communication: when to communicate and what data to send, e.g. film rendering threads may hold 2 frames: 1 being processed and another ready to go as soon as the current frame is complete

Synchronisation: ensuring threads can cooperate without interference, e.g. if threads representing 2 players compete to get a single resource then only one succeeds.

Coordination Abstraction Levels

You have probably used many notations for specifying, designing & constructing **computations** but relatively few for coordination.

Computations can be written in languages with different **levels of abstraction**, e.g.

Low-level	**Mid-level**	**High-Level**
Assembler	Java	Haskell, SQL
C	Python	Prolog

Coordination Abstraction Levels

You have probably used many notations for specifying, designing & constructing **computations** but relatively few for coordination.

Computations can be written in languages with different **levels of abstraction**, e.g.

Low-level	**Mid-level**	**High-Level**
Assembler	Java	Haskell, SQL
C	Python	Prolog

Likewise coordination can be written in languages with different levels of abstraction, e.g.

Low-level	**Mid-level**	**High-Level**
Locks,	Go	OpenMP
Semaphores	Monitors(Java threads)	
	C++ Threads	Erlang

Concurrent Coordination options for Languages

A programming language may have **several concurrency options**, often competing. For example

C++ has:

- **Thread libraries,** e.g. POSIX
- **std threads**
- …

Java has:

- **Thread libraries,** e.g. POSIX
- **Java threads**
- **Executors**
- …

University of Glasgow

Higher Level Concurrency

In the remainder of the course we will

- Cover some widely used, and higher-level, concurrency constructs in C and C++
- Illustrate how higher-level constructs can be implemented using lower-level constructs.
  - the bounded buffer example above shows a monitor implemented using mutexes & condition variables

University of Glasgow

Semaphores

- Another classical synchronization primitive, also invented by by *Edsger Dijkstra*
- A (counting) semaphore holds an integer counter and provides 2 operations:

  wait decrements the counter and blocks if it is less than zero

  signal increments the counter and wakes waiting threads
- We can count how many items of a resource are used and block access if no resources are available

University of Glasgow

Semaphores

- A binary semaphore (with a count of 1)
  - combines a mutex and a condition variable
  - Allows a single thread into a critical section: blocking/reawakening threads
- We can use a pair of semaphores to ensure a correct usage of an unsafe bounded buffer implementation

University of Glasgow

Bounded Buffer Semaphore Implementation

```
struct BoundedBuffer { int start; int end; int size; int* buffer; };
typedef struct BoundedBuffer BoundedBuffer;

sem_t fillCount; // data in the buffer
sem_t emptyCount; // free space in the buffer

BoundedBuffer * createBoundedBuffer(int size) { ... }
void destroyBoundedBuffer(BoundedBuffer * bb) { ... }
void addItem(BoundedBuffer * bb, int item) { ... }
int removeItem(BoundedBuffer * bb) { ... }
void * producer(void * arg) { ... }
void * consumer(void * arg) { ... }

int main() {
  pthread_t t1; pthread_t t2; int err;
  BoundedBuffer * bb = createBoundedBuffer(4);
  fillCount = sem_create(0, 4); // no data in the buffer yet
  emptyCount = sem_create(4, 4); // all spaces in the buffer are free
  err = pthread_create(&t1, NULL, consumer, bb); assert(err == 0);
  err = pthread_create(&t2, NULL, producer, bb); assert(err == 0);
  err = pthread_join(t1, NULL); assert(err == 0);}
```

University of Glasgow

Bounded Buffer Semaphore: Producer

```
void * producer(void * arg) {
  BoundedBuffer * bb = (BoundedBuffer*)arg;
  for (int i = 0; i < 10; i++) {
    sem_wait(emptyCount);
    int item = randInt();
    printf("produced item %d\n", item);
    addItem(bb, item);
    sem_signal(fillCount);
    usleep(randInt());
  }
  return NULL;
}
```

University of Glasgow

Semaphores are simpler than locks & cond. vars

```
int removeItem(BoundedBuffer * bb) {
  if (!bb) assert(0);
  pthread_mutex_lock(&bb->m);                                    sem_wait(emptyCount);
  while ( ((bb->end + 1) % bb->size) == bb->start ) {
    printf("== Buffer is empty ==\n");
    pthread_cond_wait(&bb->remove, &bb->m);
  }
  // buffer is no longer empty
  bb->end = (bb->end + 1) % bb->size; // move end one forward
  int item = bb->buffer[bb->end];
  pthread_mutex_unlock(&bb->m);                                  sem_signal(fillCount);
  pthread_cond_signal(&bb->add);
  return item;
}
```

University of Glasgow

Bounded Buffer Semaphore: Consumer

```
void * consumer(void * arg) {
  BoundedBuffer * bb = (BoundedBuffer*)arg;
  for (int i = 0; i < 10; i++) {
    sem_wait(fillCount);
    int item = removeItem(bb);
    printf(" consumed item %d\n", item);
    sem_signal(emptyCount);
    usleep(randInt());
  }
  return NULL;
}
```

University of Glasgow

Semaphore Implementation

- Uses a combination of a mutex and a condition variable:

```
typedef struct sem {
   pthread_mutex_t mut; // mutex to protect value
   pthread_cond_t cv; // condition variable to signal change to value
   unsigned int value; // the current value of the counter
   unsigned int maxval; // the maximum value of the counter
} Semaphore;
```

- wait decrements the counter and blocks if it goes below zero:

```
void sem_wait (Semaphore* s) {
   pthread_mutex_lock(&(p->mut));
   while (p->value <= 0) { pthread_cond_wait(&(p->cv), &(p->mut)); }
   p->value--;
   pthread_mutex_unlock(&(p->mut)); }
```

- signal increments the counter and signals waiting threads:

```
void sem_signal(Semaphore* s) {
  pthread_mutex_lock(&(p->mut));
  p->value++;
  pthread_cond_signal(&(p->cv));
```

University of Glasgow

Concurrency Reflection

- Safely and correctly managing concurrent threads that share state is tricky
- You must avoid problems of
  - Deadlock
  - Livelock: threads run, but make no progress
  - Starvation: some thread never makes progress
- If you use locks you must correctly lock, unlock, and wait to arrange safe access

University of Glasgow

To Do Now

In the “Examples from Lecture 6” folder on Moodle

- Review the code for the semaphore bounded buffer:

  bounded_buffer_sem.c

- Compile and run the program a couple of times & reflect on the results

University of Glasgow

Systems Programming
C++ Threading
Phil Trinder

University of Glasgow

Review of Lecture 6

Introducing Concurrency

- We learned that **concurrency** is a programming paradigm to deal with lots of things at once
- This is distinct from **parallelism** which is about making a program run faster
- **Threads** execute concurrently inside of the same **process** and share the same **address space**
- Communication between threads uses **shared memory** and requires careful synchronisation

Low Level Concurrency

- We learned how to create and wait for a thread using pthreads in C
- **Race conditions** can be avoided using *mutual exclusion* which we ensure with a mutex/lock
- **Condition variables** avoid *busy waiting* for a condition to become true

University of Glasgow

Review of Lecture 6

Higher Level Concurrency

- We distinguished coordination & computation, and learned that concurrency can be specified at different levels of abstraction
- We can design data structures like a **bounded buffer** that internally use synchronisation mechanisms to ensure *thread-safe* usage: **Monitor**s
- Counting/binary Semaphores are higher level, combining mutex, condition variable & a counter

University of Glasgow

This lecture: higher level concurrency in C++

- C++ provides
  - low-level pthread constructs
  - a mid-level thread implementation as part of the standard library (since C++ 2011)
  - higher-level threading constructs
- We are going to learn some C++ features that are particularly useful for programming with threads

Hello World from a C++ thread

```
#include <stdio.h>
#include <thread>
int main() {
  auto t = std::thread([](){
    printf("Hello World from a cpp thread!\n");
  });
  t.join();
}
```

- Q: what happens if we leave out the t.join()?
- We need to `#include <thread>` and compile with `-std=c++11` (or greater) to specify the C++ standard:

```
% clang++ -std=c++17 -stdlib=libc++ -pthreads
    -o hello_world hello_world.cpp
```

- There are three major "modern" C++ standards: c++11, c++14, and c++17
- Pick the latest supported by your compiler!
- In the example we use two features of C++ that we haven't seen so far: *auto* and *lambdas*

University of Glasgow

auto: Local Type Inference

In C++ it is possible (and encouraged) to use the auto keyword instead of the name of the type when declaring variables:

```
auto i = 42; // means the same as: int i = 42;
```

The compiler will *infer* (i.e. figure out) the type of the variable from the initialisation expression

Crucially the variable still has a data type and the compiler knows it, we just do not need to write it down

This feature becomes in particular handy for long type names, e.g.:

```
auto v = std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8, 9};
// using explicit type name:
for (std::vector<int>::iterator iter = v.begin(); iter != v.end(); iter++) { /*...*/ }
/ / using auto instead:
for (auto iter = v.begin(); iter != v.end(); iter++) { /*...*/ }
```

University of Glasgow

Lambda expressions

We have seen a *function pointer used* to pass a function as an argument,
e.g. to create threads with pthread:

```
void * PrintHelloWorld(void* arg) { printf("Hello World\n"); }
int main() {
  // ...
  int error = pthread_create(&thread, NULL, PrintHelloWorld, NULL);
                                          // ... ^ this is passed as a function pointer
}
```

By writing a *lambda expression* we can write a function without naming it:

```
int main() { auto t = std::thread( [](){ printf("Hello World\n"); } ); t.join(); }
```

Other names for *lambdas* are: *function literal* or *anonymous functions*
Lambda expressions refer to the *lambda calculus* invented by Alonzo Church in the 1930s
They are available in almost all modern programming languages

Lambda expressions C++ syntax

Lambda expressions in C++ are written as:

```
[ /*captures*/ ] ( /*params*/ ) { /*body*/ }
```

The *captures* lists variables from the surrounding scope that are passed to the lambda *when it is created*

The *params* list the parameters that are passed to the lambda *when it is called*

The *body* is the function body that executes when the lambda is called

As for function parameters variables are captured *by-value* i.e. they are copied

University of Glasgow

Sharing a Value with a Lambda expression

We can *share* access to a variable by capturing a pointer to it using this notation:

```
int main() {
  auto l = list{}; l.append_to_list('a'); l.append_to_list('b’);
  auto t1 = std::thread([l_ptr = &l] (){ l_ptr->remove_from_list(1); });
t1.join();
}
```

University of Glasgow

Threads and Lambdas

Lambda expressions are very useful for writing applications with threads
We start a thread in C++ by constructing an std::thread object with a function pointer or lambda
In pthreads function pointers are passed as void* to threads, losing the type information!

```
void* remove_second_element_from_list(void* arg) {
  struct list* list = (struct list*)arg; // restore the meaning of the pointer
  // ...
}
int main() {
  struct list* list = create_list(create_node('a')); // ...
  pthread_create(&t1, NULL, remove_second_element_from_list, list ); //
...
}
```

Lambda Expressions Preserves Function Types

Capturing values for a lambda expression preserves their types

```
int main() {
   auto l = list{}; l.append_to_list('a'); // ...
   auto t1 = std::thread([l_ptr = &l] (){ l_ptr->remove_from_list(1); }); // ...
 }
```

University of Glasgow

Mutual exclusion: remembering to unlock

Recall that

- *mutual exclusion* means that no two threads simultaneously enter a critical section
- Commonly managed with a mutex and call `lock()` before we enter and `unlock()` when we leave a critical section

But it's very easy to forget to call unlock when code has complicated control flow or exceptions

Q: What happens if some control flow fails to unlock a mutex?
Q: What happens if some control flow fails to lock a mutex?

University of Glasgow

Mutual exclusion in C++

C++ avoids the issue of forgetting to unlock a resource by viewing locking mutex as *owning a resource* and applying the RAII (Resource Acquisition Is Initialization) technique:
We create a local variable on the stack that locks the mutex
at the end of the lifetime the variable releases the lock *automatically*

```
#include <mutex>
std::mutex m; // mutex variable; shared across threads
void foo() {
  std::unique_lock<std::mutex> lock(m); // acquire the mutex
  // ... do some work
} // releases the mutex by calling m.unlock();
```

University of Glasgow

Thread Safe Linked List in C++

We encapsulate an std::list<int> and add a thread-safe interface protected by a (private) std::mutex

```
#include <list> #include <thread> #include <optional> #include <mutex>
struct list {
private:
  std::list<int> list; std::mutex mutex;    // mutex to protect critical section
public:
  void append_to_list(int value) {
    std::unique_lock<std::mutex> lock(mutex); // lock mutex: enter critical section
    list.push_back(value);
  }                                          // mutex will be automatically unlocked here
std::optional<int> remove_from_list(int position) {
  std::unique_lock<std::mutex> lock(mutex); // lock mutex: enter critical section
  auto iter = list.begin();
  while (position > 0 && iter != list.end()) { iter++; position--; }
  if (position != 0 || iter == list.end()) { return {}; /* nothing to return */ }
  int value = *iter;
  list.erase(iter);
  return value;
 }                                          // mutex will be automatically unlocked here
};
```

University of Glasgow

Thread Safe Linked List in C++

```
std::list<int>::iterator begin()  { return list.begin(); }
std::list<int>::iterator end()    { return list.end(); }
}

 int main() {
   auto l = list{}; l.append_to_list('a'); l.append_to_list('b'); l.append_to_list('c');
   auto t1 = std::thread([l_ptr = &l](){ l_ptr->remove_from_list(1); });
   auto t2 = std::thread([l_ptr = &l](){ l_ptr->remove_from_list(1); });
   t1.join(); t2.join();
 }
```

Q: Why don't we need to lock the mutext for the begin() and end() methods?

Q&A Demo: Unsafe linked list

University of Glasgow

To Do Now

Starting with the “Examples from Lecture 7” folder on Moodle

1. Review the code for `cppthreads_hello_world.cpp`
   Compile it and run it
2. Review the code for `cppthreads_hello_world_bad.cpp`
   Compile it and run it. Write a sentence explaining what went wrong, and how to fix it.
3. Review the code for `linked_list_safe.cpp`
   Compile it and run it
4. Can you make the linked list implementation that is unsafe? That is an implementation that allows the threads to interfere and demonstrates a race condition?
   Hint: there are different ways to do this, and I’ll demo one in the Q&A
   Hint: you’ll probably need a larger list and more threads

University of Glasgow

Condition variables in C++

Condition variables are a synchronisation mechanism to wait for conditions without busy waiting

For pthreads we learned to always use a while loop when using a condition variable:

```
pthread_mutex_lock(&m);
while (!cond) {
  pthread_cond_wait(&cv, &m); }
```

In C++ we directly provide the test for the condition we are waiting for to the wait call:

```
cv.wait(m, [](){ return cond; });
```

This idiom checks the condition after the thread is woken up, guaranteeing that cond is true after the statement is executed.

University of Glasgow

Condition variables in C++

Condition variables are a synchronisation mechanism to wait for conditions without busy waiting

For pthreads we learned to always use a while loop when using a condition variable:

```
pthread_mutex_lock(&m);
while (!cond) {
  pthread_cond_wait(&cv, &m); }
```

In C++ we directly provide the test for the condition we are waiting for to the wait call:

```
cv.wait(m, [](){ return cond; });
```

This idiom checks the condition after the thread is woken up, guaranteeing that cond is true after the statement is executed.

Be aware about the opposite checks of cond here:

- In C pthread the while checks if the condition is *not* true an may repeatedly wait
- In C++ we write the check the other way around to express that we wait for the condition to become true

University of Glasgow

Bounded Buffer in C++

The C++ implementation is simpler than C with pthreads:

- We only need to provide a thread safe wrapper for an std::vector
- The std::vector automatically takes care of the memory management

```
struct BoundedBuffer {
private:
  int start; int end; int size;
  std::vector<int> buffer; // use std::vector<int> for automatic memory management
  std::mutex m;
  std::condition_variable add_cv;          // there is space to add an item to the buffer
  std::condition_variable remove_cv;       // there is an item in the buffer that can be removed
public:
  BoundedBuffer(int max_size) { ... } // constructor to create bounded buffer
  void addItem(int item) { ... } // thread-safe interface to add items to the buffer
  int removeItem() { ... } // thread-safe interface to remove items from the buffer
};
  int main() {
  auto bb = BoundedBuffer{4}; // create a bounded buffer object
  auto consumer = std::thread([bb = &bb]{ ... }); // start consumer thread
  auto producer = std::thread([bb = &bb]{ ... }); // start producer thread
  consumer.join(); producer.join(); // wait for both threads to finish
}
```

University of Glasgow

Bounded Buffer in C++ Producer

```
int main() {
  ...
  // start producer thread and capture pointer to the shared bounded buffer
  auto producer = std::thread([bb = &bb]{
    for (int i = 0; i < 10; i++) {
      int item = randInt();
      printf("produced item %d\n", item);
      bb->addItem(item);
      std::this_thread::sleep_for(std::chrono::milliseconds(randInt()));
    }
  });
  consumer.join(); producer.join(); // wait for both threads to finish
}
```

The consumer implementation is similar

University of Glasgow

Bounded Buffer in C++ Constructor

```
struct BoundedBuffer {
private:
  int start; int end; int size;
  std::vector<int> buffer; // use vector for automatic memory management
  std::mutex m;
  std::condition_variable add_cv;
  std::condition_variable remove_cv;
public:
  BoundedBuffer(int max_size) { // constructor to create bounded buffer
    start = 0;
    end = max_size-1;
    size = max_size;
    // no manual malloc (and free) required when using std::vector
    buffer = std::vector<int>(size);
  }
  void addItem(int item) { ... } // thread-safe interface to add items to the buffer
  int removeItem() { ... } // thread-safe interface to remove items from the buffer
};
```

University of Glasgow

Bounded Buffer in C++ addItem

```
struct BoundedBuffer {
private:
  ...
public:
  BoundedBuffer(int max_size) { ... }

  void addItem(int item) { // thread-safe interface to add items to the buffer
    std::unique_lock<std::mutex> lock(m); // acquire lock: enter critical section
    add_cv.wait(lock, // wait with the condition variable 'add_cv'
       [this]{ return start != end; }); // for the cond 'start != end' to become true
    buffer[start] = item;
    start = (start + 1) % size;
    remove_cv.notify_one(); // notify possibly waiting thread that an item is in the buffer
  } // lifetime of 'lock' reached; release lock: exit critical section
  ...
};
```

University of Glasgow

Bounded Buffer in C++ removeItem

```
struct BoundedBuffer {
private:
  ...
public:
  BoundedBuffer(int max_size) { ... }

  int removeItem() { // thread-safe interface to remove items from the buffer
    std::unique_lock<std::mutex> lock(m); // acquire lock: enter critical section
    remove_cv.wait(lock, // wait with the condition variable 'remove_cv'
      [this]{ return ((end + 1) % size) != start; }); // for this condition to become true
    end = (end + 1) % size;
    int item = buffer[end];
    add_cv.notify_one(); // notify waiting threads that there is free space in the buffer
    return item;
  } // lifetime of 'lock' reached; release lock: exit critical section  ...
};
```

University of Glasgow

To Do Now

Starting with the “Examples from Lecture 7” folder on Moodle

1. Review the code for for the bounded buffer with condition variables:
   `bounded_bufer_safe.cpp`
   Compile it and run it a few times

University of Glasgow

Asynchronous Tasks

- So far we have created threads explicitly and used low-level coordination
- Higher-level abstractions are easier to use and harder to get wrong
- Instead of threads we can think about *tasks* that should be handled concurrently
- Instead of synchronisation we can think about how these tasks *communicate*
- C++ provides a high-level interface to launch an asynchronous task: std::async …

University of Glasgow

Launching tasks with std::async

```
int fib(int n) { ... } // computes the nth fibonacci number

int main() {
  // launch task to computer fibonacci number of 6
  auto f6 = std::async([] { return fib(6); });
  // while we are waiting for the task to finish we compute the fibonacci number of 7
  auto f7 = fib(7);
  printf("fib(7) = %d\n", f7);
  // now we access the result of the task and wait if it is not yet finished computing
  printf("fib(6) = %d (computed asynchronously)\n", f6.get() );
}
```

University of Glasgow

Futures std::future

A task launched with std::async returns an std::future:

```
int main() {
  std::future<int> f6 = std::async([] { return fib(6); });
  // ...
  printf("fib(6) = %d (computed asynchronously)\n", f6.get() );
}
```

A *future* is a handle representing a value that is not yet computed

University of Glasgow

Futures std::future

We can pass a future around or store it somewhere, for example

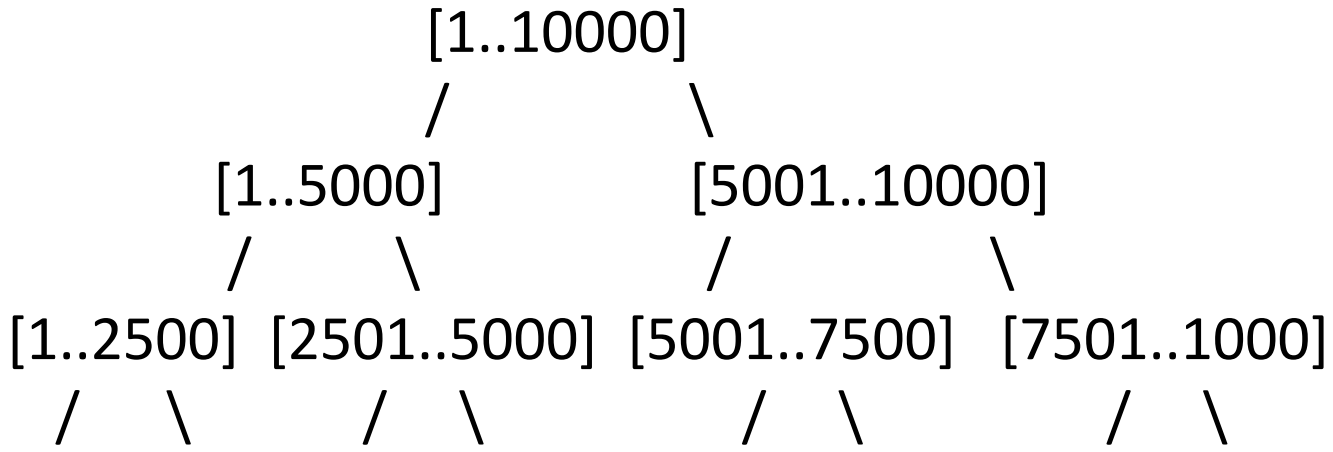
```
int main() {
  auto fs = std::vector<std::future<int>> ();
  // launch 10 asynchronous tasks to compute fibonacci numbers
  for (int i = 0; i < 10; i++) { fs.push_back(std::async([] { return fib(i+1); })); }
   // ... do some other work; then print the computed numbers
   for (int i = 0; i < 10; i++) { printf("fib(%d) = %d\n", i+1, fs[i].get()); }
 }
```

Call future.get() to retrieve the value. Blocks calling thread until the value has been computed

University of Glasgow

Futures Example: parallel sum

Reduce the time to compute the sum of all values in an array by using multiple threads
Use divide & conquer by recursively splitting the array in half, and creating a thread to compute each half, e.g.



Conceptually we could keep dividing until we are summing arrays that in contain 0 or 1 element, but these threads would hardly take any time at all …
So it's sensible to set a threshold, e.g. not divide intervals containing fewer than 1000 elements, instead compute these sequentially in a thread

University of Glasgow

Futures Example: parallel sum

```
int parallelSum(std::vector<int>::iterator begin, std::vector<int>::iterator end) {
  auto len = end - begin;
  // compute sequentially for small arrays
  if (len < 1000) { return std::accumulate(begin, end, 0); }
  auto mid = begin + len/2;
  // launch asynchronous task for the left half of the array
  auto left_side = std::async([=] { return parallelSum(begin, mid); });
  // compute right half of array recursively
  int right_side = parallelSum(mid, end);
  // block to wait for left side to finish
  return left_side.get() + right_side;
}

int main() {
  std::vector<int> vec = createLargeVector();
  auto sum = parallelSum(vec.begin(), vec.end());
  printf("sum: %d\n", sum);
}
```

University of Glasgow

Parallel sum Performance

We run the program on an array with 10^6 elements

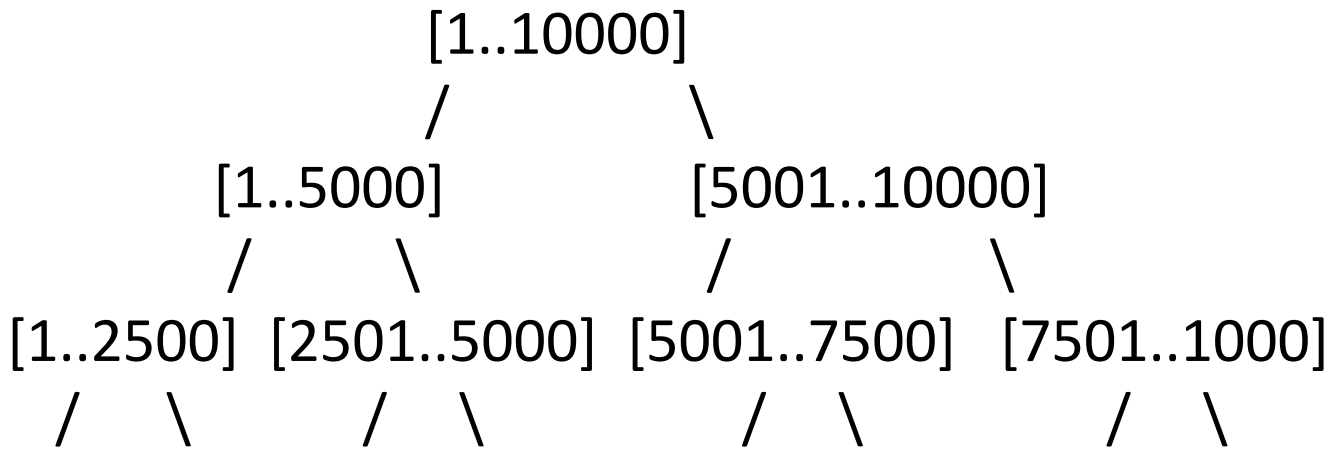
Sequential sum result: 5243777 (time: 5.491102 ms)

Parallel par_sum result: 5243777 (time: 135.894128 ms)

The parallel version is about 30 times slower than the sequential version!

Why?

parallel sum thread analysis



Q: How many threads are created **at** depth 3?

Q: How many threads are created **at** depth *d*?

Q: How many threads are created to sum the values in a 10^6 element array, with threshold 1000?

University of Glasgow

A better parallel sum

The naïve parallel sum creates too many threads, and the time managing all of these threads outweighs the time saved by running the threads

If we have a small number of cores (say 8) we can fix this, by only spawning threads for the first few recursive calls:

```
int parallelSum(std::vector<int>::iterator begin, std::vector<int>::iterator end, int depth = 0 )
{
  auto len = end - begin;
  if (len < 1000 || depth > 3 ) { return std::accumulate(begin, end, 0); }
  auto mid = begin + len/2;
  auto left_side = std::async([=] { return parallelSum(begin, mid, depth + 1 ); });
  int right_side = parallelSum(mid, end, depth + 1 );
  return left_side.get() + right_side;
}
```

Resulting in a much improved parallel runtime:

```
sum result: 5243777 (time: 4.979168 ms) vs. par_sum result: 5243777 (time: 2.492056 ms)
```

University of Glasgow

To Do Now

Starting with the “Examples from Lecture 7” folder on Moodle

1. Review the code for the naive parallel sum using asynchronous tasks :
   `simple_sum.cpp`
   Compile it and run it
2. Review the code for the parallel sum with a depth threshold:
   `sum.cpp`
   Compile it, run it
   What do you conclude from the runtimes?
3. Review the code for the bounded buffer using asynchronous tasks:
   `bounded_bufer_safe_async.cpp`
   Compile it and run it a few times

University of Glasgow

Communicating with an async task: std::promise

A task created with async communicates its result via a future

Sometimes we want to name the channel that the task should write to: a promise

We can see a *future* as the reading end of a communication channel:
some_future.get() extracts the value

The writing end of the channel is called a *promise:*
some_promise.set_value(42);

We can obtain a std::future object from a std::promise:

```
std::future<int> some_future = some_promise.get_future();
```

University of Glasgow

Using a promise

```
// This sum function writes to the sum_promise
void sum(std::vector<int>::iterator begin,
         std::vector<int>::iterator end,
         std::promise<int> sum_promise) {
  int sum = std::accumulate(begin, end, 0);
  sum_promise.set_value(sum); // 4. write result
}

int main() {
  auto numbers = std::vector<int>{ 1, 2, 3, 4, 5, 6 };
  std::promise<int> sum_promise; // 1. create promise for an int
  std::future<int> sum_future = sum_promise.get_future(); // 2. get future from promise
  // 3. create thread that takes ownership of the promise (ownership transfer with std::move)
  auto t = std::thread(sum, numbers.begin(), numbers.end(), std::move(sum_promise) );
  printf("result = %d\n", sum_future.get() ); // 4. wait and then read result
  t.join();
}
```

University of Glasgow

Synchronisation with std::promise<void>

Usually we communicate data over a channel, e.g. the results of a computation
Sometimes we only want to synchronise tasks e.g. wait until all worker threads are ready before sending work
We can achieve this with
- an std::promise<void>, a *promise* to produce *nothing* (but say when you're done)
- the std::future<T>::wait() method

```
void do_work(std::promise<void> barrier) {
 std::this_thread::sleep_for(std::chrono::seconds(1)); // do something (like sleeping)
 barrier.set_value(); // 4. send signal to other thread
}
int main() {
 std::promise<void> barrier; // 1. create promise
 std::future<void> barrier_future = barrier.get_future(); // 2. get future from it
 auto t = std::thread(do_work, std::move(barrier) ); // 3. launch thread
 barrier_future.wait(); // 4. wait for signal
   …                    // 5. continue execution: we know that do_work has completed
 t.join(); }
```

University of Glasgow

Tasks as First Class Objects

Most programming languages aim to treat values of all types as **first class**, e.g. you can use a value of any type anywhere, e.g. pass it as a parameter, store it in a data structure.

Q: How true is this of languages you know? Can you pass as a parameter, store in a data structure, return from functions: integers, strings, structs/objects, functions?

We've seen that we can treat communication channels (aka promises/futures) as first class in C++, but what about tasks?

Manipulating tasks is essential, e.g. to write a thread scheduler, perhaps for a virtual machine.

University of Glasgow

Manipulating tasks with std::packaged_task

We can

- wrap up a task for future execution using std::packaged_task
- extract the value with task.get_future()

```
int main() {
  auto task = std::packaged_task<int(int,int)>([](int a, int b) { return pow(a, b); }) ;
  std::future<int> result = task.get_future() ;

    // The task can now be stored, or passed as a parameter.
    // When we are ready to use it either
    // launch task in the same thread via:
    // task(2, 10);
    // or start a new thread:
  auto t = std::thread(std::move(task) , 2, 10);
  t.join();
  printf("task result: %d\n", result.get() );
}
```

University of Glasgow

To Do Now

Starting with the “Examples from Lecture 7” folder on Moodle

1. Review and run `promise.cpp`
   What happens if you call `sum_future.get()` a second time?
   Adapt the program to print the sum twice.

2. Review and run `promise_void.cpp`
   Adapt the main function to wait for the barrier signal, i.e. call `barrier_future.wait()` a second time

3. Review and run `packaged_task.cpp`
   Adapt the program to evaluate `task(3,8)` without creating a thread
   What happens if the main function evaluates task a second time, say with `task(4,7)`? Explain why.