Computer Science for HEP

Module 4

Fabio Cufino IMAPP

C++ stuff that I should already know

Now I'll start a little recap of the thing that I would like not to forget.

Objects

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects

- An object is a region of storage (i.e. memory)
  - it has a type
  - it has a lifetime
  - it can have a name

A type identifies a set of values and the operations that can be applied to those values
- C++ is a strongly typed language (mostly)

- Arithmetic types
  - integral types
    - signed integer types: short int, int, long int, long long int
    - unsigned integer types: unsigned short int, unsigned int, unsigned long int, unsigned long long int
    - character types: char, signed char, unsigned char, . . .
    - boolean types: bool
  - floating-point types: float, double, long double
- std::nullptr_t
  - type of the null pointer nullptr
- void
  - denotes absence of type information

int

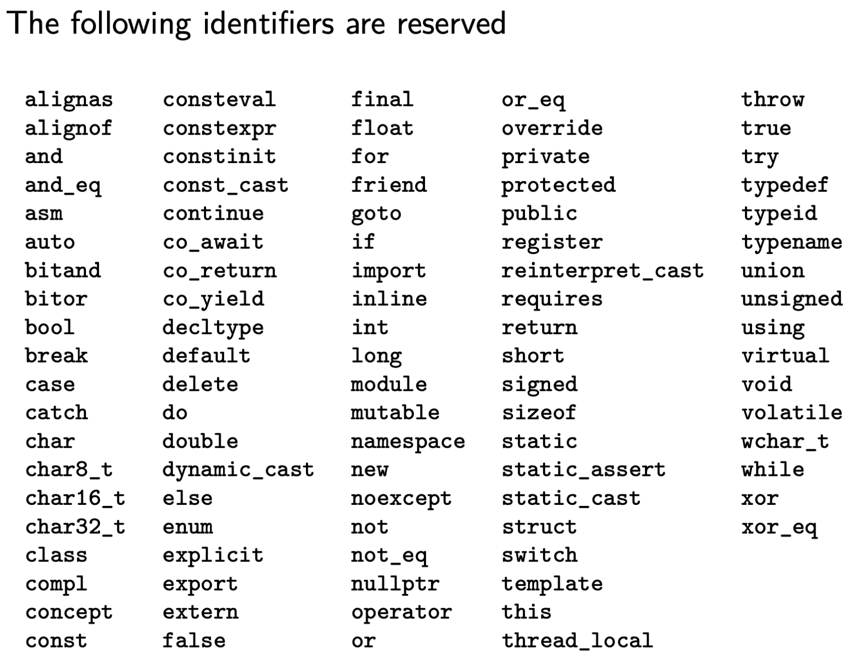
- With N bits, values are in the range $(-2^{N-1}, 2^{N-1}-1)$
- Pay attention to the negative numbers rapresentation

7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

- Typical size is 32 bits (4 bytes)

Identifiers

- An identifier is a sequence of letters (including `_`) and digits, starting with a letter
- Identifiers are used to name entities in a program



The following identifiers are reserved

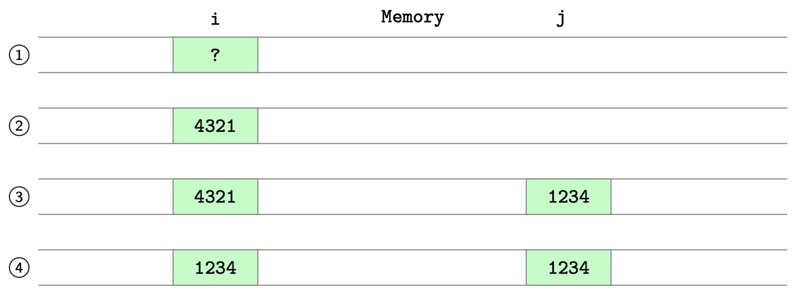
alignas	consteval	final	or_eq	throw
alignof	constexpr	float	override	true
and	constinit	for	private	try
and_eq	const_cast	friend	protected	typedef
asm	continue	goto	public	typeid
auto	co_await	if	register	typename
bitand	co_return	import	reinterpret_cast	union
bitor	co_yield	inline	requires	unsigned
bool	decltype	int	return	using
break	default	long	short	virtual
case	delete	module	signed	void
catch	do	mutable	sizeof	volatile
char	double	namespace	static	wchar_t
char8_t	dynamic_cast	new	static_assert	while
char16_t	else	noexcept	static_cast	xor
char32_t	enum	not	struct	xor_eq
class	explicit	not_eq	switch	
compl	export	nullptr	template	
concept	extern	operator	this	
const	false	or	thread_local	

-

Variables

- A variable is an identifier that gives a name to an object

```
int i; // declaration; the value is undefined
i = 4321; // assignment of a constant
int j{1234}; // declaration and initialization in one step
i = j; // assignment of j's value to i
```



Literals

A literal is a constant value of a certain type included in the source code:
• integer • floating point • character • string • boolean • null pointer • user-defined

std::string

- Provided by the C++ Standard Library
- A compound (user-defined) type to represent a string of characters
- An `std::string` can be initialized with a string literal, a sequence of escaped or non-escaped characters between double quotes
  - "hello" "hello'\n'world" "hello "world""
  - \n means “newline”
- The type of a string literal is not std::string

```
std::string corso{"Programmazione per la Fisica"};
corso = corso + "\nAnno Accademico 2023/2024"; \\WRONG
```

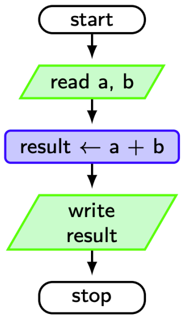
Flow Control

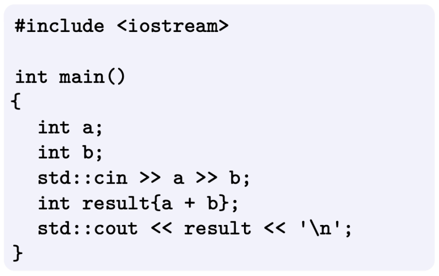
Algorithms

A finite sequence of precisely defined steps to solve a problem

Statements

Statements are units of code that are executed in sequence:
• expression statement • compound statement or block • declaration statement • selection statement • iteration statement • jump statement





```
#include <iostream>

int main()
{
  int a;
  int b;
  std::cin >> a >> b;
  int result{a + b};
  std::cout << result << '\n';
}
```

Expression statement

- An expression followed by a semicolon (;)
- An expression is a sequence of operators and their operands that specifies a computation
- The evaluation of an expression typically produces a result

Block

A sequence of zero or more statements enclosed between braces ({})

Declaration statement

- A declaration statement introduces one or more new identifiers into a C++ program, possibly initializing them
  ◦ typically variables, but not only

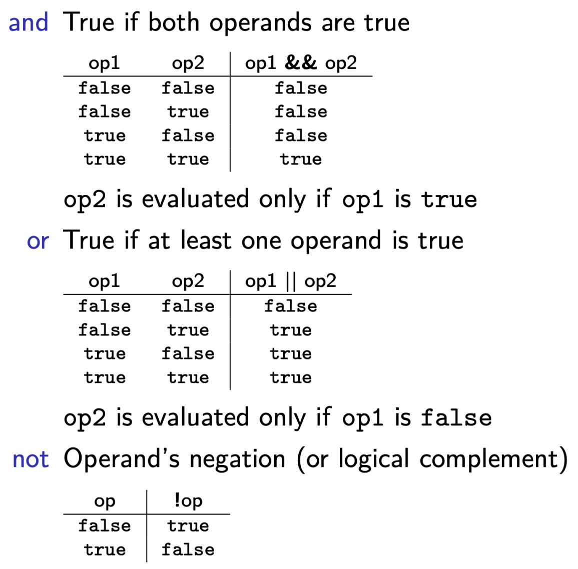
- A declaration of a variable in a block makes the variable of automatic storage duration, unless otherwise specified
    - the corresponding object is automatically created each time the declaration is executed
    - the corresponding object is automatically destroyed each time the execution reaches the end of the block
- A declaration should introduce only one identifier
- A variable should be declared only in the moment it's actually needed
- A variable should be initialized at the point of declaration
    - There are very few exceptions, if any, to this recommendation

Scope

The scope of a name appearing in a program is the, possibly discontiguous, portion of source code where that name is valid

---

Logical operations



and True if both operands are true

op1	op2	op1 && op2
false	false	false
false	true	false
true	false	false
true	true	true

op2 is evaluated only if op1 is true

or True if at least one operand is true

op1	op2	op1 || op2
false	false	false
false	true	true
true	false	true
true	true	true

op2 is evaluated only if op1 is false

not Operand's negation (or logical complement)

op	!op
false	true
true	false

Double

64 bits, smallest values ≈±10^{-308}, largest values ≈±10^{308}

- Precision is about 16 decimal digits

Float

32 bits, smallest values ≈±10^{-38}, largest values ≈±10^{38}

- Precision is about 7 decimal digits

Standard mathematical functions

`cmath`
The cmath header includes many ready-to-use mathematical

functions

- Exponential
- Power
- Trigonometric
- Interpolation
- Hyperbolic
- Floating-point manipulation, classification and comparison

```
#include <cmath>

double x{⋯};
std::sqrt(x);
std::pow(x, .5);
std::sin(x);
std::log(x);
std::abs(x);
```

Type conversions

A value of type T1 may be converted implicitly to a value of type T2 in order to match the expected type in a certain situation

- Conversions can be explicit using `static_cast`

```
1 + static_cast<int>(2.3)
```

Mechanism exist to define implicit and explicit conversions involving user-defined types

const-safety

Data qualified as const is logically immutable. Data that is meant to be immutable should be const.

```
int const x{1'000'000'000}; // or const int
std::cout << x + 32; // ok, read-only
x += 32; // error, trying to modify
int const y; // error, not initialized and not modifiable later
```

```
std::string const message{"Hello"};
std::cout << message + " Francesco"; // ok, read-only
message += " Francesco"; // error, trying to modify
std::string const empty_message; // ok! empty string
```

Functions

- A function declaration contains the essential information needed to invoke the function `return-type function-name(parameter-list);
- If the declaration is followed by the actual block of statements (the implementation of the function), it is also a definition return-type function-name (parameter-list) {···}
    - Note the block scope
- Each parameter in the parameter list is of the form type name_opt
    - type is mandatory
    - name is optional
        - in the declaration, but useful for documentation purposes
        - in the definition if it's not used
- If the function returns nothing, the return type is void

```
int isqrt(int); // declaration
int count_words(std::string s) {··· } // definition
double pow(double base, double exp); // declaration
void print(std::string); // declaration
int generate_random_number() {··· } // definition
```

- Within the function block, the return statement returns the result (and the control) to the calling function
- For a function returning a non-void type return expression ;
- The result of expression must be convertible to the return type
- For a function returning void
  return;
- At the end of the function, return; is optional.

A function needs to be declared/defined before it's used

Recursive functions

A function can call itself, directly or indirectly

- called recursion
- Often an elegant alternative to a loop

```
int sum_n(int n)
{
        // assume n >= 0
        if (n == 0) { // base case
        return 0;
        } else { // recursive case
          return n + sum_n(n - 1);
        }
}
```

Function overloading

Multiple functions can share the same name but must have different parameter lists (number and/or types).

- **Compiler Behavior:**
  - The compiler selects the function that best matches the call, applying implicit conversions when appropriate.
  - Errors occur if no match or multiple equally valid matches exist.
- **Return Type:**
  - The return type is irrelevant for function overloading.

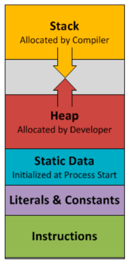
  ```
  void foo(int);
  int foo(int, char);
  bool foo(double);
  int foo(std::string s);
  ```

  - `foo(0);` → Calls `foo(int)`
  - `foo(0, '0');` → Calls `foo(int, char)`
  - `foo(0.);` → Calls `foo(double)`
  - `foo(std::string{});` → Calls `foo(std::string)`
  - `foo(0L);` → Ambiguous, error
  - `foo('a');` → Calls `foo(int)`
  - `foo("a");` → Calls `foo(std::string)`

Memory layout of a process

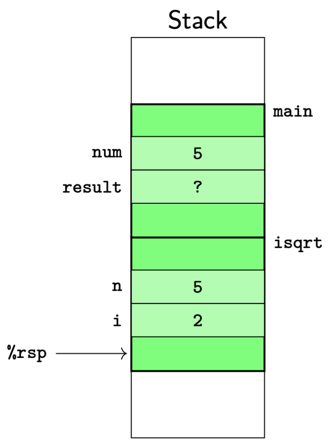
- A process is a running program
- When a program is started the operating system brings the contents of the corresponding file into memory according to well-defined conventions

- Stack
  - function local variables
  - function call bookkeeping
- Heap
  - dynamic allocation
- Global data
  - literals and variables
  - initialized and uninitialized (set to 0)
- Program instructions



Functions and the stack

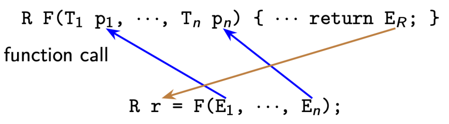
```
int isqrt(int n)
{
  int i{1};
  while (i * i < n) {
    ++i;
  }
  if (i * i > n) {
    --i;
  }
  return i;
}

int main()
{
  int num;
  std::cin >> num;
  int result{isqrt(num)};
  std::cout << result << '\n';
}
```



The state of the stack just before returning from isqrt

Pass by Value, Return by Value

Given a function



$$\mathtt{R\ F(T_1\ p_1,\ \cdots,\ T_n\ p_n)\ \{\ \cdots\ return\ E_R;\ \}}$$

and a function call

$$\mathtt{R\ r\ =\ F(E_1,\ \cdots,\ E_n);}$$

- Each p_i is initialized with the value of expression E_i
  - Every time the function is called a new p_i is created, which gets destroyed at the end of the function
  - NB if E_i is just a variable, p_i is another object (a copy) and changing it inside the function doesn't change the original object corresponding to the variable
- r is initialized with the value of expression E_R

Example:

```
int add(int a, int b) {
    return a + b; // `a` and `b` are copies of the arguments passed
}

int main() {
    int x = 5, y = 10;
    int sum = add(x, y); // `x` and `y` are passed by value
    // Modifying `a` or `b` in `add` has no effect on `x` or `y`
    return 0;
}
```

Hene:

- **Pass-by-value:** Arguments are copied into function parameters.
- **Return-by-value:** The return expression is copied to the caller’s variable.
- Changes inside the function do not affect the original variables. This provides isolation and ensures the integrity of the original arguments.

Passing by value may be inconvenient

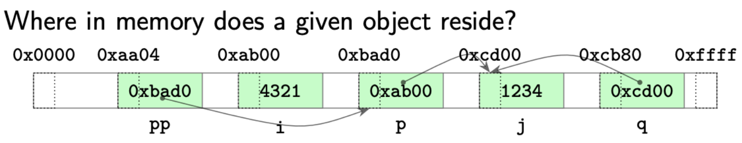
Consider a function that increments an int object

```
void increment(??? n) {
  // ++n
}

int number{42};
increment(number);
// number == 43
```

- NB The function does not return the new value; it modifies the passed object in place
- We cannot write void increment(int n), because the function would modify a copy of the original object

Where in memory does a given object reside?

```
int i{4321};
int j{1234};
std::cout << &i; // 0xab00, address-of operator
std::cout << &j; // 0xcd00
int* p{&i}; // pointer declarator
std::cout << &p; // 0xbad0
int** pp{&p}; // &p is of type int**
p = &j;
int* q{p}; // p and q point to the same object
```



Pointers and References in C++

1. Introduction to Pointers

Pointers are variables that store the memory address of another variable.

Declaring a Pointer

To declare a pointer, use the `*` symbol:

```
int num = 10;
int *ptr = &num; // 'ptr' now holds the address of 'num'
```

Accessing the Value via Pointer

To access the value at the address a pointer holds, use the dereference operator `*`:

```
std::cout << *ptr; // Output: 10
```

Changing Value through a Pointer

You can modify the variable's value by dereferencing the pointer:

```
*ptr = 20;
std::cout << num; // Output: 20
```

Passing a pointer

```
void increment(int* n) {
  ++(*n);
}

int number{42};
increment(&number);
// number == 43
```

```
int count_words(std::string* s)
{
  int count{0};
  ··· *s ···
  return count;
}

std::string text{···};
count_words(&text);
```

- The caller takes the address of the object and passes it to the function
- The function dereferences the pointer to get access to the object
- Be careful not to dereference a null pointer

Const and pointers

```
std::string text{···};
std::string* ptext{&text};        // ok, can read/modify name via *ptext
std::string const* cptext{&text}; // ok, can only read text via *cptext
```

```
std::string const text{···};
std::string* ptext{&text};        // error, else could modify text via *ptext
std::string const* cptext{&text}; // ok, can only read text via *cptext
```

2. Introduction to References

References are another name for an existing variable. They cannot be null or reassigned after initialization.

Declaring a Reference

Use the `&` symbol to declare a reference:

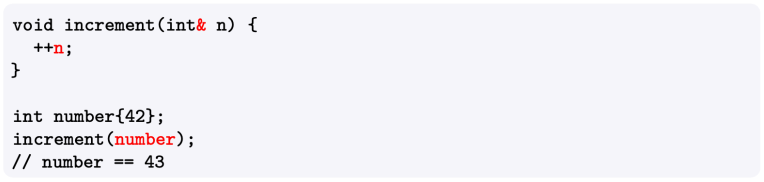
```
int num = 10;
int &ref = num; // 'ref' is a reference to 'num'
```

Accessing and Modifying Values

Accessing or modifying the reference affects the original variable:

```
std::cout << ref; // Output: 10
ref = 20;
std::cout << num; // Output: 20
```

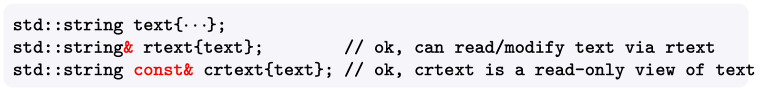
Passing bt reference



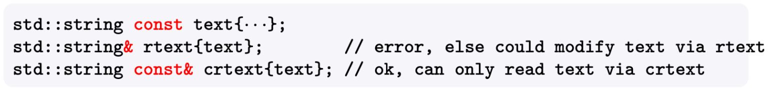
```
void increment(int& n) {
  ++n;
}

int number{42};
increment(number);
// number == 43
```

- There is no difference in the caller compared to pass-by-value
- There is no difference in the body of the function compared to pass-by-value
- The only visual clue is in the parameter declaration

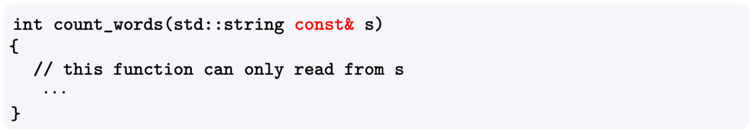
Const and reference



```
std::string text{···};
std::string& rtext{text};         // ok, can read/modify text via rtext
std::string const& crtext{text}; // ok, crtext is a read-only view of text
```



```
std::string const text{···};
std::string& rtext{text};         // error, else could modify text via rtext
std::string const& crtext{text}; // ok, can only read text via crtext
```



```
int count_words(std::string const& s)
{
  // this function can only read from s
   ···
}
```

3. Key Differences Between Pointers and References

Aspect	Pointer	Reference
Nullability	Can be null	Cannot be null
Reassignment	Can point to different variables	Must be initialized and cannot be reassigned
Syntax	`*` for dereferencing	Access like a regular variable

4. Example of Using Both in a Function

In this example, we'll use pointers and references to modify values inside a function.

```
#include <iostream>

void updateWithPointer(int *ptr) {
    *ptr += 5; // Modifies the value at the memory address
}

void updateWithReference(int &ref) {
    ref += 5; // Directly modifies the value of the referenced variable
}

int main() {
    int num1 = 10, num2 = 10;

    updateWithPointer(&num1);    // Passing by pointer
    updateWithReference(num2);    // Passing by reference

    std::cout << "After updateWithPointer: " << num1 << std::endl; // Output: 15
    std::cout << "After updateWithReference: " << num2 << std::endl; // Output: 15

    return 0;
}
```

In this example:

- `updateWithPointer` takes a pointer, `*ptr`, to modify `num1`.
- `updateWithReference` takes a reference, `ref`, to modify `num2`.

How to pass arguments to functions

- For input parameters
  - If the type is primitive, pass by value

```
int isqrt(int n);        // good
int isqrt(int const& n); // bad! pessimization
```

- Otherwise pass by const reference

```
int count_words(std::string const&);
```

- For output parameters, prefer to return a value or pass by non-const reference

```
int read_from_cin();      // may be more difficult to overload for other types
void read_from_cin(int& n);
```

- For input-output parameters, pass by non-const reference

```
void to_lowercase(std::string& s);
```

Returning a reference

A function can return a reference only if the referenced object survives the end of the function
◦ Otherwise, in the caller, the reference would refer to an object that doesn't exist anymore

In particular do not return a reference to a function local variable

```
// bad
int& add(int a, int b) {
  int result{a + b};
  return result;
}
```

```
// acceptable
int& increment(int& a) {
  ++a;
  return a;
}
```

Useful to chain multiple function calls on the same object

```
std::string& to_lower(std::string& s) { · · · ; return s; }
std::string& trim_right(std::string& s) { · · · ; return s; }
std::string& trim_left(std::string& s) { · · · ; return s; }

std::string s{· · ·};
trim_left(trim_right(tolower(s)));
```

range-for loop

for (*range-declaration* : *range-expression*) *statement*

- Simplified form of a for loop, to iterate on all the elements of a range (sequence), such as a string of characters
- Execute repeatedly statement for all the elements of the range
- range-declaration declares a variable of the same type of an element of the range
  - Can (and should) be a (const) reference
- range-expression represents the range to iterate over

```
std::string s{"Hello!"};
for (char& c : s) {
 c = std::toupper(c);
}

for (int i : {1, 2, 3, 4, 5}) {
 std::cout << i << ' ';
}
```

Auto

Let the compiler deduce the type of a variable from the initializer, i.e. from the expression used to initialize the object

- auto never deduces a reference
- auto preserves constness of references

Enumerations

An **enumeration** is a distinct type with named constants called **enumerators**.
2. **Scoped Enumerations (`enum class`):**
- Enumerators are specified with the name of the enumeration and the scope-resolution operator (`::`), e.g., `Operator::Plus`.
- Default values:
- The first enumerator has the value `0` by default.
- Subsequent enumerators increment by 1 unless explicitly set.
- Explicit values can be assigned, e.g., `Plus = -2`.
3. **Underlying Type**:

- By default, the underlying type is `int`.
- It can be changed, e.g., `enum class byte : unsigned char { };`.
- Values of the underlying type are valid enumeration values.
4. **Type Conversion**:
- Conversion to the underlying type requires explicit casting, e.g., `int i = static_cast<int>(Operator::Plus);`.
5. **Unscoped Enumerations**:
- No `class` keyword, e.g., `enum Operator { Plus, Minus };`.
- Enumerator symbols are visible in the enclosing scope (no need for `::`).
- Implicit conversion to the underlying type is allowed.
- Use scoped enumerations for better type safety and clarity.

The switch statement

The switch statement transfers control to one of multiple statements, depending on the value of a condition

```
double compute(char op, double left, double right) {
double result;

switch (op) {

        case '+': result = left + right;
        break;

        . . .

        case '/': result = (right != 0.) ? left / right : 0.;
        break;

 default: result = 0.;
 }

return result;
}
```

The condition is an expression whose evaluation gives an integral or enumeration value

- Cannot switch on strings, for example
- Typically each statement is followed by a break statement, to jump after the switch, otherwise control falls through the next instruction, even if this is part of a statement introduced by another labe

Data abstraction

The C++ language has a strong focus on building lightweight data abstractions

- The source code can use terminology and notation close to the problem domain, making it more expressive
- There is little (if any) overhead in terms of space or time during execution
- Class and struct are the primary mechanism to define new compound types on top of fundamental types.

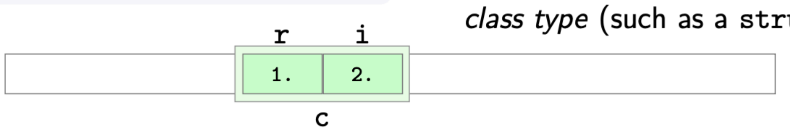
Let's introduce a type to represent complex numbers

- We can pass/return Complex objects to/from functions, take pointers and references, . . .

```
struct Complex {
  double r; // data member (field)
  double i;
};

double norm2(Complex c);
Complex sqrt(Complex c);

Complex c{···};
double n{ norm2(c) };
Complex c2{ sqrt(c) };

Complex& cr{c};  // reference
Complex* cp{&c}; // pointer
```

```
double norm2(Complex c) {
  return c.r * c.r + c.i * c.i;
}

double norm2(Complex const& c) {
  return c.r * c.r + c.i * c.i;
}
```

- A Complex is composed of two doubles
- The . (dot) operator allows to access a member of an object of *class type* (such as a struct)



- It's possible to define operations on user-defined types

```
struct Complex {
  double r;
  double i;
};

bool operator==(Complex const& a, Complex const& b) {
  return a.r == b.r && a.i == b.i;
}

Complex operator+(Complex const& a, Complex const& b) {
  return Complex{a.r + b.r, a.i + b.i};
}

c2 = c1       // generated by the compiler, if used
c1 == c2
c1 + c2
z = z * z + c
2. * c1
...
```

Private and public

Imagine to change the Complex type to use the polar form

```
struct Complex { double rho; double theta; };
```

As a consequence, all client code has to change

```
double norm2(Complex const& c) { return c.rho * c.rho; }
```

Changing the internal representation (e.g., from Cartesian to polar coordinates in `struct Complex`) requires modifying all client code.
- Some combinations of data (e.g., (ρ,θ) with invalid ρ<0 θ not in (0,2π) might not be valid.

Desired Solution:
- Increase isolation between client code and internal representation.
- Enforce internal relationships (class invariants) between data members.

Encapsulation

The internal representation of data should be considered an implementation detail and hidden from clients.

Manipulation of objects should be done via a **well-defined function-based interface**.
2. **Implementation:**
- Declare data members (`x`, `i`) as **private**.
- Provide **public member functions** (methods) for interaction.
3. **Naming Convention:**
- Use special naming conventions for data members (e.g., `_suffix` or `m_prefix`).

Example:

```
#include <stdexcept> // For exceptions

class Complex {
        private:
          double r_;
          double i_;

        public:
                ...constructor...

            void setReal(double real) {
                if (real < 0) {
                    throw std::invalid_argument("Real part cannot be negative");
                }
                r_ = r;
                }
};
```

Construction

A class can have a special function, called **constructor**, which is called to initialize the storage of an object when it is created.

- It should initialize all data members, in order to establish the class invariant

Best Practices:
- Use the **member initialization list** to initialize data members in the order they are declared.

Example:

```
        Complex(double x, double y) : x{x}, i{y} { }
```

- Avoid performing additional logic in the constructor body if it can be done in the initialization list.
- Data members are initialized in the order of declaration and should preferably be initialized in the member initialization list
- The constructor's name is the same as the class name

```
class Complex {
        private:
          double r_;
          double i_;
        public:
          Complex(double x, double y) // no return type
          : r_{x}, i_{y} // member initialization list
          { /* nothing else to do */ }

...
};

Complex c{1., 2.}; // or (1., 2.)
```

- For a class, private is the default and can be omitted
- Member functions that don't modify the object should be declared `const`
- A class can have multiple constructors
- A data member can be given a default initializer, which is used if the member is not explicitly initialized in the called constructor

```
class Complex {
        double r_{0.};
        double i_{0.};
public:
        Complex(double x, double y) : r_{x}, i_{y} {}
        Complex(double x) : r_{x} {} // i_ initialized with 0
        Complex() = default; // r_ and i_ initialized with 0
...
};

Complex c1{1., 2.};
Complex c2{1.}; // meaning {1., 0.}
Complex c3; // meaning {0., 0.}
```

Pointers and data structures

- address-of operator: `&`
◦ Given an object it returns its address in memory

- dereference operator: `*`
◦ Given a pointer to an object it returns a reference to that object

- structure dereference operator: `->`
◦ Given a pointer to an object of class/struct type, it returns a reference to a member of that object

```
struct S {
        int n;
        void f();
};

S q{···};
S* p = &q;

p->n; // equivalent to (*p).n
p->f(); // equivalent to (*p).f()
```

Exceptions

Exceptions provide a general mechanism to:

- notify the occurrence of an error in the program execution,

typically when a function is not able to accomplish its task, i.e. to satisfy its post-condition

- using a throw expression

```
class Rational
{
private:
        int n_;
        int d_;

public:
        Rational(int num = 0, int den = 1) : n_{num}, d_{den}

{

if (d_ == 0) {
        throw std::invalid_argument("Denominator cannot be zero");
}
```

An exception is an object

- After being raised (thrown), an exception is propagated up the stack of function calls until a suitable catch clause (handler) is found
- If no suitable handler (i.e. one compatible with the type of the exception) is found, the program is terminated

```
auto function1() {

try {
        ··· // this part is executed
        throw E{};
        ··· // this part is not executed

} catch (E const& e) {
        ··· // use e -> // manage the error, e.g. log a message on stdandard error
}

}
```

- An exception is typically raised by a constructor to inform that it is not able to properly initialize the object

Templates

Let's consider again the Complex class

```
class Complex {
        double r;
        double i;

public:
        Complex(double x = double{}, double y = double{}) : r{x}, i{y} {}
                double real() const { return r; }
                double imag() const { return i; }
        };
```

What if we want float members?

```
class Complex {
  double r;
  double i;
 public:
  Complex(
      double x = double{}
    , double y = double{}
  ) : r{x}, i{y} {}
  double real() const { return r; }
  double imag() const { return i; }
};
```

```
class Complex {
  float r;
  float i;
 public:
  Complex(
      float x = float{}
    , float y = float{}
  ) : r{x}, i{y} {}
  float real() const { return r; }
  float imag() const { return i; }
};
```

We can transform Complex into a template, with the floating-point type a parameter of that template

```
template<typename FP> // or, template<class FP>

class Complex {

        private:
                FP r;
                FP i;

        public:
                Complex(FP x = FP{}, FP y = FP{}) : r{x}, i{y} {}
                FP real() const { return r; }
                FP imag() const { return i; }
};
```

Now we also have to specify the FP

```
Complex c; // error
Complex<double> d; // instantiation of a Complex<double> type
Complex e{1.,1.}; // Complex<double> deduced
```

The C++ Standard Library

How can we guarantee that in a program composed of thousands of files, written by thousands of people, using third-party libraries, there are no conflicts between identifiers?

- Namespaces are a mechanism to partition the space of names in a program to prevent such conflicts

```
// in <vector>

namespace std {
        template<class T> vector {··· };
}
```

The standard library contains components of general use

◦ containers (data structures)
◦ algorithms
◦ strings
◦ input/output
◦ mathematical functions
◦ random numbers
◦ regular expressions
◦ concurrency and parallelism
◦ filesystem

Containers of objects

A program often needs to manage collections of objects
◦ e.g. a string of characters, a dictionary of words, a list of particles, a matrix, ...

- A container is an object that contains other objects

std::vectorT

- Dynamic container of elements of type T
- its size can vary at runtime
- **layout is contiguous in memory**

```
#include <vector>

std::vector<int> a;      // empty vector of ints
std::vector<int> b{2};   // one element, initialized to 2
std::vector<int> c(2);   // two elements (!), value-initialized (0 for int)
std::vector<int> d{2,1}; // two elements, initialized to 2 and 1
std::vector<int> e(2,1); // two elements, both initialized to 1


auto f = b; // make a copy, f and b are two distinct objects
f == b;     // true
```

- The size method gives the number of elements in the vector
- The empty method tells if the vector is empty
- operator[] gives access to the ith element
- The push_back method adds an element at the end of the vector

```
vec.push_back(-2); // vec is now {4,5,7,-2}
vec.push_back(0); // vec is now {4,5,7,-2,0}
```

Iterators and ranges

An iterator is an object that indicates a position within a range

- A container, such as a vector, is a range
  Ranges are typically obtained from containers calling methods begin and end

```
std::vector<int> v {···};
auto first = v.begin(); // std::vector<int>::iterator
auto last = v.end(); // std::vector<int>::iterator
```

- Syntactically, operations on iterators are inspired by pointers

---

Erese

- The erase method removes the element pointed to by the iterator passed as argument (must not be end())

```
// remove the central element
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it);
```

- The erase method can remove a range, passing two iterators

```
// erase the 2nd half of the vector
auto it = v.begin() + v.size() / 2; // iterator to the middle element
v.erase(it, v.end()); // size ~halved here
```

In general, iterators pointing to erased elements are not valid anymore

◦ For a vector, iterators pointing to elements following the erased one are also invalidated, including the end iterator

```
v.erase(it);
*it; // undefined behaviour (UB)
++it; // UB, be careful in loops
it =···; // ok
```

Insert

The insert method inserts an element before the position indicated by an iterator

```
// insert the value 42 in the middle of the vector
auto it = v.begin() + v.size() / 2;
v.insert(it, 42); // size increased by 1
```

Array(N,T)

> `std::array<T, N>`

```
#include <array>

// 2 ints, uninitialized
```

```
std::array<int,2> a;

// 2 ints, initialized to 1 and 2
std::array<int,2> b{1,2};

// 2 ints, value-initialized (0 for int)
std::array<int,2> c{};

// 2 ints, initialized to 1 and 0
std::array<int,2> d{1};

// make a copy
auto e = b;
assert(e == b);
```

- he size method gives the number of elements in the array (which corresponds to N)
- operator[] gives access to the ith element
- begin, end, empty, front, back methods
- No push_back or insert, the size is fixed.

Algorithms

- Generic functions that operate on ranges of objects
- Implemented as function templates

Example:

- Sum all the elements of a container cont of ints

```
int sum {0};

for (auto it = cont.begin(), last = cont.end(); it != last; ++it) {
        sum += *it;
}
```

This is better:

```
auto sum = std::accumulate(cont.begin(), cont.end(), 0); // better
```

Find an element equal to val in a container cont

```
auto it = cont.begin();
auto const last = cont.end();

for (; it != last; ++it) {
        if (*it == val) {
                break;
        }
}
```

This is better:

```
auto it = std::find(cont.begin(), cont.end(), val); // better
```

Hierarchy of iterators

```
// InputIterator (find)
while (first != last && !(*first == value)) ++first;
```

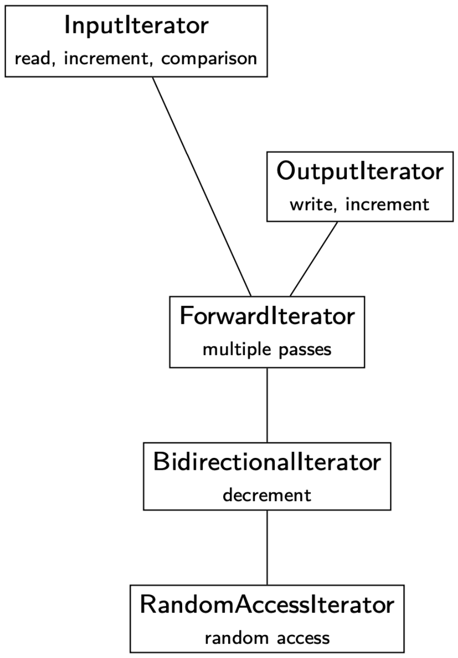
```
// OutputIterator (generate_n)
for (; n > 0; ++first, --n) *first = gen();
```

```
// ForwardIterator (generate)
for (; first != last; ++first) *first = gen();
```

```
// ForwardIterator (adjacent_find)
auto i = first;
while (++i != last) ...
```

```
// BidirectionalIterator (reverse)
if (first == --last) break;
```

```
// RandomAccessIterator (reverse)
for (; first < --last; ++first) ...
```



Examples of algorithms

Non-modifying	all_of any_of for_each count count_if mismatch equal find find_if adjacent_find search ...
Modifying	copy copy_if fill generate transform remove replace swap reverse rotate shuffle sample unique ...
Partitioning	partition stable_partition ...
Sorting	sort partial_sort nth_element ...
Set	includes set_union set_intersection ...
Min/Max	min max minmax clamp ...
Comparison	equal lexicographical_compare ...
Numeric	iota accumulate inner_product partial_sum adjacent_difference reduce ...

Algorithms in action:

```
std::array a {23, 54, 41, 0, 18};

// sort the array in ascending order
```

```
std::sort(std::begin(a), std::end(a));

// sum up the array elements, initializing the sum to 0
auto s = std::accumulate(std::begin(a), std::end(a), 0);

// append the partial sums of the array elements into a vector
std::vector<int> v;
std::partial_sum(std::begin(a), std::end(a), std::back_inserter(v));

auto p = std::inner_product(std::begin(a), std::end(a), std::begin(v), 0);

// find the first element with value 42, if existing
auto it = std::find(std::begin(v), std::end(v), 42);
```

Why using standard algorithms

- They are correct
- They express intent more clearly than a raw for/while loop
- They are efficient
  - They give computational complexity guarantees
  - How fast do they run? how much additional memory do they need?
- They enable easy access to parallelism

Algorithms and functions

Notes with Code Snippets

- find **Algorithm:**

```
template <class Iterator, class T>

Iterator find(Iterator first, Iterator last, const T& value) {
    for (; first != last; ++first)
        if (*first == value)
            break;
    return first;
}

auto it = find(v.begin(), v.end(), 42);
```

  - Searches for 42 in the range [v.begin(), v.end()).
  - Returns an iterator to the found element or v.end() if not found.

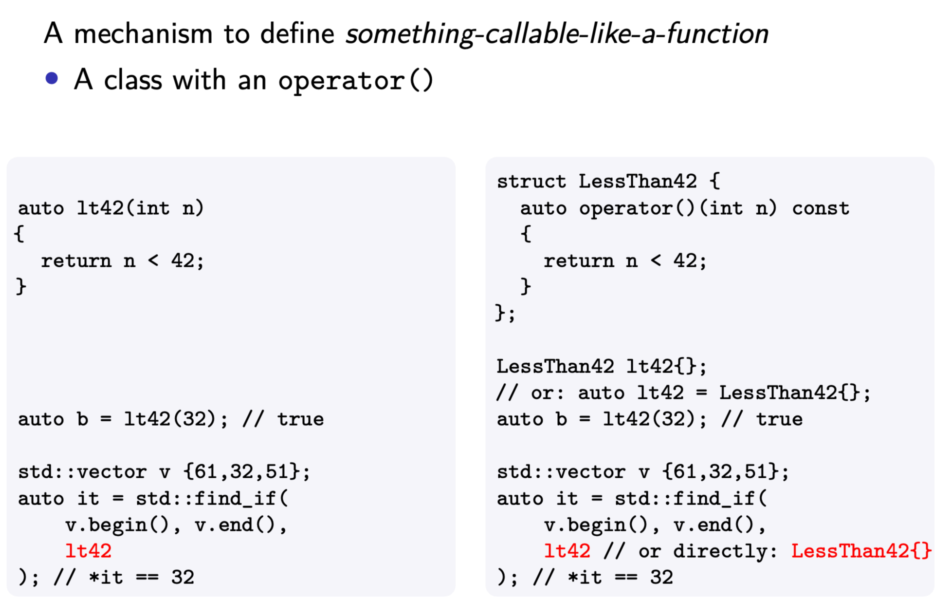
find_if **Algorithm:**

```
template <class Iterator, class Predicate>

Iterator find_if(Iterator first, Iterator last, Predicate pred) {
    for (; first != last; ++first)
        if (pred(*first)) // unary predicate
            break;
    return first;
}

bool lt42(int n) { return n < 42; }
auto it = find_if(v.begin(), v.end(), lt42);
```

- Uses a predicate lt42 to search for the first value less than 42.

Algorithms like find and find_if are versatile for searches in ranges. Functions or lambdas enable custom logic within algorithms for flexibility.

Function objects



A mechanism to define *something-callable-like-a-function*

- A class with an operator()

```
auto lt42(int n)
{
  return n < 42;
}




auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42
); // *it == 32
```

```
struct LessThan42 {
  auto operator()(int n) const
  {
    return n < 42;
  }
};

LessThan42 lt42{};
// or: auto lt42 = LessThan42{};
auto b = lt42(32); // true

std::vector v {61,32,51};
auto it = std::find_if(
    v.begin(), v.end(),
    lt42 // or directly: LessThan42{}
); // *it == 32
```

- A function like lt42 is straightforward but less flexible.
- A function object (LessThan42) allows additional customization, such as maintaining state or using templates for more complex conditions.

A **function object**, being an instance of a class, can have **state**. This allows you to encapsulate additional data (state) within the function object and use it in the callable operator().

Example Breakdown:

1. Class Definition:

```
class LessThan {
    int m_; // The state (threshold value for comparison)
public:
    explicit LessThan(int m) : m_{m} {} // Constructor to initialize the state
    auto operator()(int n) const {
        return n < m_; // Callable function using the stored state
    }
};
```

• m: *This is a member variable that stores the threshold value.*
• ***Constructor**: Accepts an integer m to initialize the state.*
• *operator(): Takes an integer n as input and compares it with the stored threshold m.*

2. Example Usage:

a. Creating Stateful Instances:

```
LessThan lt42{42}; // State: m_ = 42
auto b1 = lt42(32); // true, because 32 < 42
// Alternative syntax: auto b1 = LessThan{42}(32);


LessThan lt24{24}; // State: m_ = 24
auto b2 = lt24(32); // false, because 32 >= 24
// Alternative syntax: auto b2 = LessThan{24}(32);
```

• Here, lt42 and lt24 are separate objects with different states (m = *42 and m =* 24 respectively).
• They behave differently depending on their state when called with the same input (32).

b. Using in Algorithms:

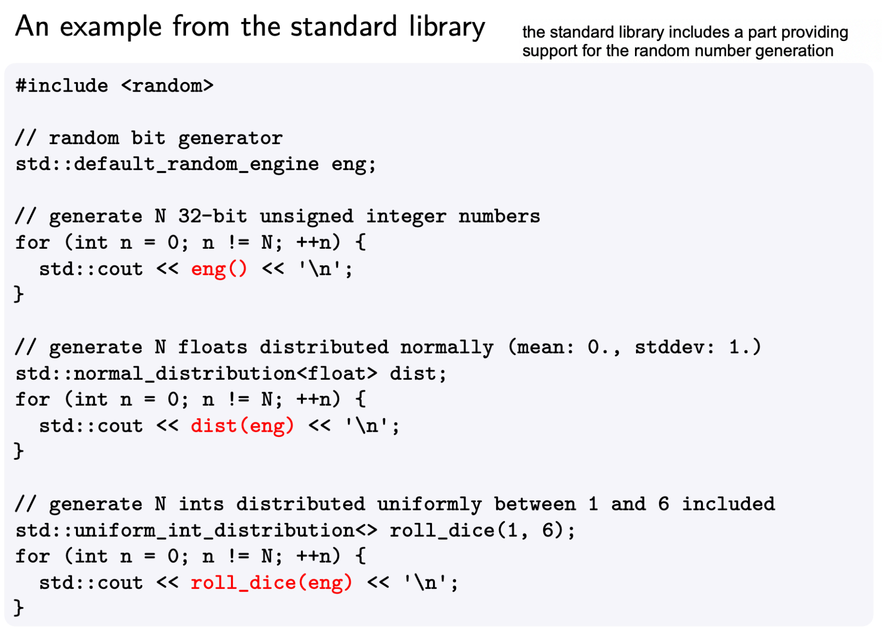
```
std::vector v {61, 32, 51};

// Search using lt42 (threshold 42)
auto i1 = std::find_if(v.begin(), v.end(), lt42);
// *i1 == 32, because 32 is the first element < 42

// Search using lt24 (threshold 24)
auto i2 = std::find_if(v.begin(), v.end(), lt24);
// i2 == v.end(), because no element is < 24
```

• lt42: The first element satisfying n < 42 is 32, so *i1 == 32.
• lt24: No element satisfies n < 24, so the iterator i2 equals v.end().

Key Takeaways:

- The LessThan class is stateful because each instance stores a threshold value (m_).
- Different instances (lt42, lt24) can have different behaviors based on their states.
- Function objects like LessThan can be directly passed to STL algorithms (e.g., std::find_if), allowing for concise and expressive code.

Here an example form the standard library



An example from the standard library

the standard library includes a part providing support for the random number generation

```
#include <random>

// random bit generator
std::default_random_engine eng;

// generate N 32-bit unsigned integer numbers
for (int n = 0; n != N; ++n) {
  std::cout << eng() << '\n';
}

// generate N floats distributed normally (mean: 0., stddev: 1.)
std::normal_distribution<float> dist;
for (int n = 0; n != N; ++n) {
  std::cout << dist(eng) << '\n';
}

// generate N ints distributed uniformly between 1 and 6 included
std::uniform_int_distribution<> roll_dice(1, 6);
for (int n = 0; n != N; ++n) {
  std::cout << roll_dice(eng) << '\n';
}
```

Lambda functions

C++ Lambda expression allows us to define anonymous function objects which can either be used inline or passed as an argument.

- They are more convenient because we don't need to overload the `()` operator in a separate class or struct.

Example:

```
#include <iostream>
using namespace std;

int main() {
  // create a lambda function that prints "Hello World!"
  auto greet = []() {
    cout << "Hello World!";
  };

  // call lambda function
  greet();

  return 0;
}
```

> Lambda Function as Argument in STL Algorithm

A concise way to create an unnamed function object.

- **Use Cases:** Ideal for actions/callbacks in algorithms, threads, or frameworks.

Example:

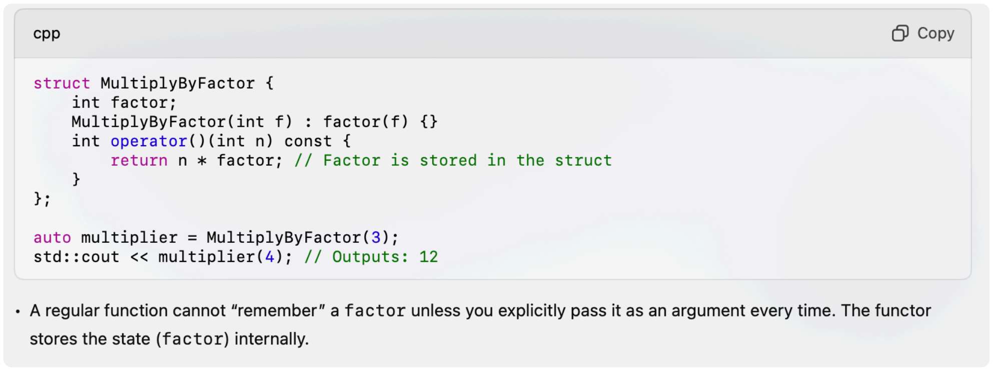
Here you understand why they are so important

```
std::find_if(container.begin(), container.end(), [](int n) {
    return n < 42;
});
```

really easy to write. Here instead the equivalent without the lambda function:

```
struct LessThan42 {
    auto operator()(int n) const {
        return n < 42;
    }
};

std::find_if(container.begin(), container.end(), LessThan42{});
```

- What is the LessThan42? Why not a simple function?



```cpp
struct MultiplyByFactor {
    int factor;
    MultiplyByFactor(int f) : factor(f) {}
    int operator()(int n) const {
        return n * factor; // Factor is stored in the struct
    }
};

auto multiplier = MultiplyByFactor(3);
std::cout << multiplier(4); // Outputs: 12
```

- A regular function cannot “remember” a `factor` unless you explicitly pass it as an argument every time. The functor stores the state (`factor`) internally.

Lambda Closure

- **Definition:** A lambda expression produces an unnamed function object, also called a **closure**.

Components:

- operator() defines the body of the lambda.
- Captured local variables become data members of the closure.

Example:

```
auto v = 42;

auto lt = [v](int n) { return n < v; };

bool result = lt(5); // true
```

Equivalent Functor Implementation:

```
class SomeUniqueName {
    int v;
```

```cpp
public:
    explicit SomeUniqueName(int v) : v(v) {}
    auto operator()(int n) const {
        return n < v;
    }
};

auto lt = SomeUniqueName{42};
bool result = lt(5); // true
```

Lambda Capture

Definition: Capturing variables from the surrounding scope in a lambda.

Capture Options:
- []: Capture nothing.
- [=]: Capture all variables by value.
- [&]: Capture all variables by reference.
- [k]: Capture k by value.
- [&k]: Capture k by reference.
- [=, &k]: Capture all by value, except k by reference.
- [&, k]: Capture all by reference, except k by value.

Example:

```cpp
int v = 3;
auto l = [&v]() { v = 5; }; // Captures v by reference
l();
std::cout << v; // Outputs: 5
```

Lambda: Const and Mutable

- **Default Behavior:** A lambda is const by default.
- Variables captured by value are not modifiable.

Mutable Lambdas: Declared with mutable to allow modification of captured variables.

- The parameter list is mandatory.
- Explicit return type (if present) comes after mutable.

Compilation model

...
Stuff that I'm supposed to know

...

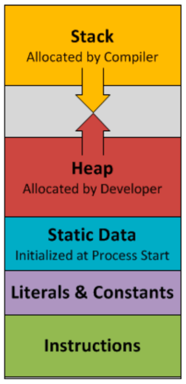
- CMake is an open-source, cross-platform family of tools designed to build, test and package software
- Build targets, dependencies, options, ...are expressed declaratively in a file called CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)
project(mandelbrot_sfml VERSION 0.1.0)
find_package(SFML 2.5 COMPONENTS graphics REQUIRED)
string(APPEND CMAKE_CXX_FLAGS " -Wall -Wextra")
add_executable(mandelbrot_sfml main.cpp)
target_link_libraries(mandelbrot_sfml PRIVATE sfml-graphics)
```

Static Data and Functions

- Objects can be created outside a function block

```
#include <random>

std::default_random_engine eng;

auto seed_rand(int s) { eng.seed(s); }
auto get_rand()       { return eng(); }
auto print_rand()     { std::cout << get_rand(); }
```



- `eng` above is a *global* variable
- They have *static* storage duration, i.e. they live for the whole program duration
- They live in the **Static Data** memory segment
- They are initialized before `main` is called and destroyed after `main` has finished

- One useful application of non-local variables is to define constants, possibly inside a namespace

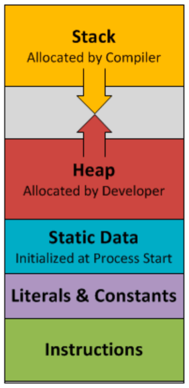
```
namespace std::numbers {
  inline constexpr double e  = ··· ;
  inline constexpr double pi = ··· ;
  ...
}
```

- `inline` means that there is only one object in the whole program (like for function definitions)
  - to be used if the definitions are in a header file
- `constexpr` guarantees that the initialization can be done at compile time
  - possibly through the execution of a *constexpr* function (not further discussed)
  - it implies `const`
- Of course constants can be defined locally as well

Explicit Memory Management

- A process is a running program
- When a program is started the operating system brings the contents of the corresponding file into memory according to well-defined conventions

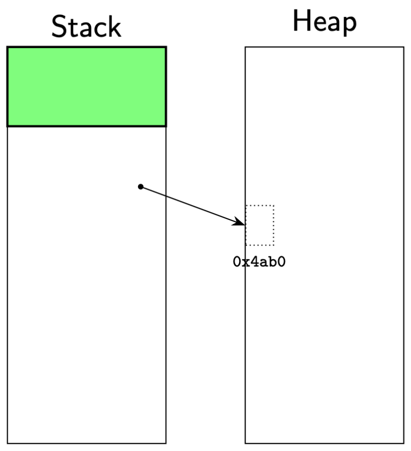
  - Stack
    - function local variables
    - function call bookkeeping
  - Heap
    - dynamic allocation
  - Global data
    - literals and variables
    - initialized and uninitialized (set to 0)
  - Program instructions

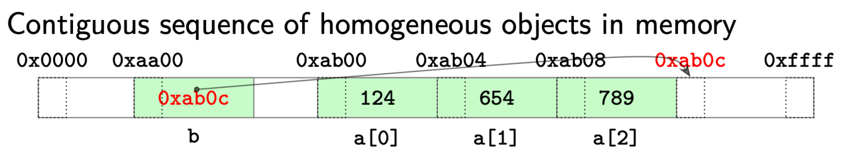


- it's not always possible or convenient to construct objects on the stack, where they would be destroyed at the end of the function that created them
- An object (array of objects) can be constructed on the free store (heap)
  - new expression (for an array: new [])
- The lifetime of an object (array of objects) on the heap is explictly managed by the developer
  - delete expression (for an array: delete [])

```
auto fun()
{
  int n {1234};
  int* p = new int{5678};
  ...
  delete p;
}
```

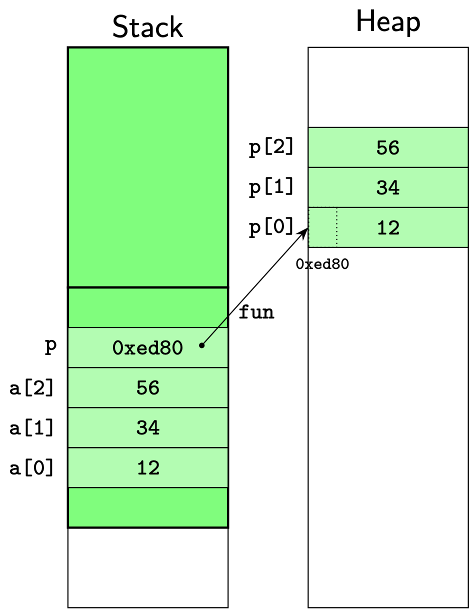
- `delete p` gives the area on the heap back to the system
- `delete p` does **not** modify `p`
- After `delete p`, the only safe operation on `p` is an assignment
  - `p = ···;`
- if `p` is `nullptr`, `delete p` is well defined and does nothing





```
int a[3] = {123, 456, 789}; // int[3], the size must be a constant
                            // and can be deduced from the initializer
++a[0];
a[3];                       // undefined behavior
// "arrays decay to pointers at the slightest provocation"
auto b = a;                 // int*, size information lost
assert(b == &a[0]);
++b;                        // increase by sizeof(int)
assert(b == &a[1]);
*b = 654;
b += 2;                     // increase by 2 * sizeof(int)
*b;                         // undefined behavior
if (b == a + 3) { ... }     // ok, but not more than this
```

```
auto fun()
{
  int a[3] {12, 34, 56};
  int* p = new int[3] {12, 34, 56};
  ...
}
```



The main function

- The main (special) function is the entry point of a program
- It can have two forms
  - int main() {···}
  - int main(int argc, char* argv[]) {···}
- If there is no return statement, an implicit return 0; is assumed
  - 0 means success, different from 0 means failure
- argc is the number of arguments on the command line, argv is an array of C-strings representing the arguments
  - argv[0] is (usually) the name of the program
  - argv[argc] is nullptr

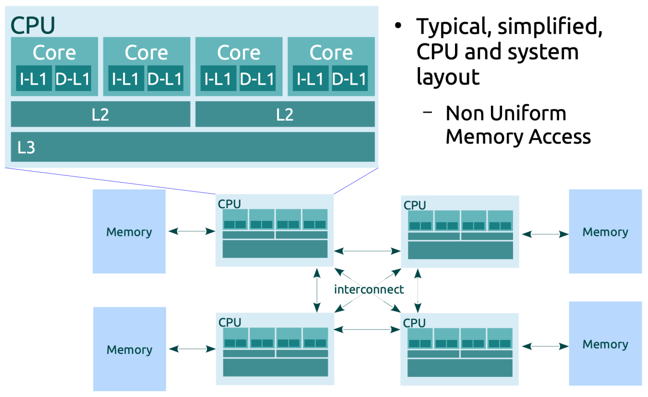
Dynamic polymorphism

Memory matters

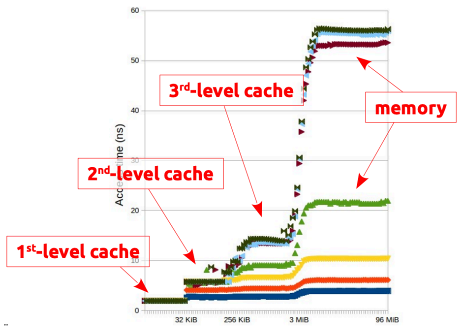
- GitHub: https://github.com/giacomini/cshep2024

Introduction

This is a typical simplified cpu and system layout


CPU
Core
I-L1 D-L1
L2
L3
Typical, simplified, CPU and system layout
Non Uniform Memory Access
Memory
CPU
interconnect

The ideal situation can be approximated with a hierarchy of different memory types. This is a plot of the access time and the various levels



- Memory is organized in a hierarchy (e.g., CPU registers, cache levels, main memory, secondary storage).
- The highest level (like CPU cache) is the fastest and closest to the processor but has less storage.
- **Hit:** When data is found in the highest level of memory (e.g., cache). This is fast, and the **hit rate** measures the fraction of accesses that are hits.
- **Miss:** When data isn't found in the highest level. The system fetches it from lower levels, which takes longer (referred to as **miss penalty**).

When a miss occurs, data is transferred from a lower level (e.g., main memory) to a higher level in blocks called **cache lines**.

Locality Principle

1. **Definition:**

- The principle states that programs tend to access memory locations in predictable patterns, which can be exploited to optimize memory hierarchy.

2. **Data Locality:**

- Example in the code (strlen function):
- The variable len is accessed multiple times, showing **temporal locality**.
- The array str is scanned sequentially, showing **spatial locality**.

Locality principle

- In a limited time interval a program accesses only a small part of its whole address space

Temporal locality

- Memory locations recently accessed tend to be accessed again in the near future
  - e.g. instructions and counters in a loop

Spatial locality

- Memory locations near those recently accessed tend to be accessed in the near future
  - e.g. sequential access to instructions in a program or to data in an array

Cache effect

The efficiency of a program does not depend only on the computational complexity of an algorithm...

- Even an algorithm with better theoretical complexity might perform worse in practice if it does not use the memory hierarchy efficiently.

Size of a type

Let's see how to help the program to have reasonable efficiency
How do you now the size in bite o the different types? `sizeof`

Type	sizeof
bool	1
char	1
short	2
int	4
long	8
long long	8
float	4
double	8
long double	16
void*	8

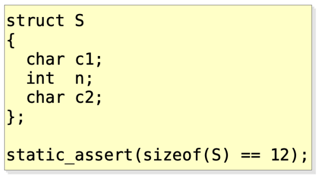
Layout of data structure

Definition: Alignment constraints mean that variables of a certain type must be stored at memory addresses that are multiples of their size.

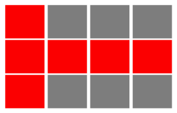
- For example, an `int` (typically 4 bytes) must be stored at addresses that are multiples of 4.

Memory access is faster when the CPU reads from addresses aligned to the type's size.
- To ensure this, the compiler often adds **padding bytes** between variables in a struct.

Example:



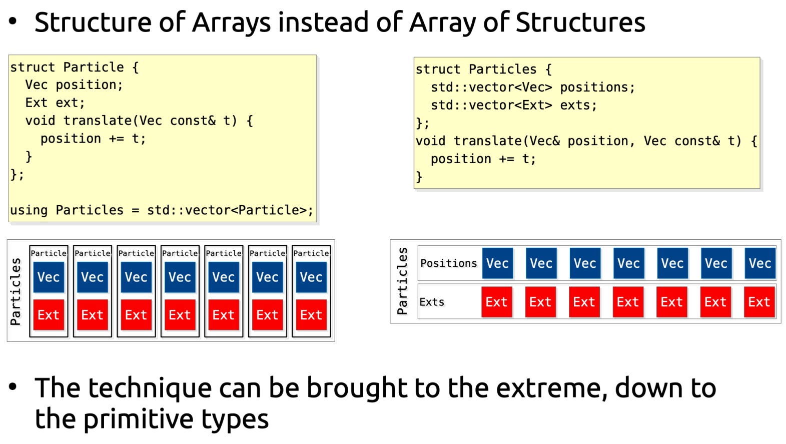
```
struct S
{
  char c1;
  int  n;
  char c2;
};

static_assert(sizeof(S) == 12);
```



- `char c1`: Takes 1 byte.
- `int n`: Takes 4 bytes but must start at a multiple of 4, so 3 padding bytes are added after `c1`.
- `char c2`: Takes 1 byte, but padding is added after it to make the struct's total size a multiple of the largest alignment (here, 4 for `int`).

Resulting layout:
- Total size is 12 bytes: 1 (c1) + 3 (padding) + 4 (n) + 1 (c2) + 3 (padding).

Alternative design techniques



- Structure of Arrays instead of Array of Structures

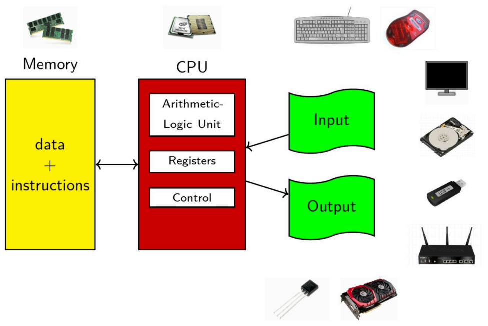
```
struct Particle {
  Vec position;
  Ext ext;
  void translate(Vec const& t) {
    position += t;
  }
};

using Particles = std::vector<Particle>;
```

```
struct Particles {
  std::vector<Vec> positions;
  std::vector<Ext> exts;
};
void translate(Vec& position, Vec const& t) {
  position += t;
}
```

- The technique can be brought to the extreme, down to the primitive types

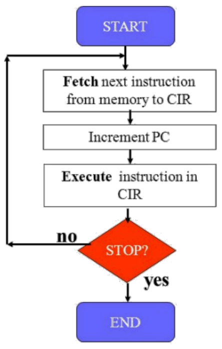
Computer architecture evolution and the performance

Von Neumann Architecture



- The basic operation that every Processing Unit (PU) has to process is called instruction and the address in memory containing the instruction is saved.
- A **Program Counter (PC)** holds the address of the next instruction
- **fetch:** the content of the memory stored at the address pointed by the PC is loaded in the Current Instruction Register (CIR) and the PC is increased to point to the next instruction's address
- **decode:** the content of the CIR is interpreted to determine the actions that need to be performed
- **execute:** an Arithmetic Logic Unit performs the decoded actions

Graphically explained:


START
Fetch next instruction from memory to CIR
Increment PC
Execute instruction in CIR
STOP?
no
yes
END

The free lunch is over

For a long time the main contribution to the gain in microprocessor performance was the increase of the clock frequency.

- Historically, **increasing clock frequency** was the primary way to improve processor performance.
- This allowed application performance to **double every 18 months** without modifying software.

End of the Trend:

- Around the mid-2000s, physical limits (like heat dissipation and power consumption) stopped the rapid increase in clock speed.
- Performance gains now require **architectural innovations** rather than simply faster processors.

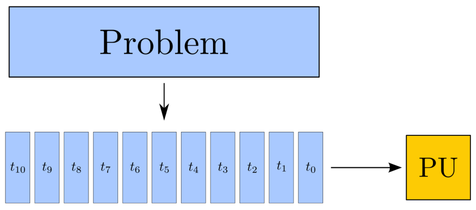
Systems evolved in different ways

- Increased number of Processing Units
- More complex control
  - Pipelining
  - Superscalar execution, hardware threading
  - Out-of-order execution, branch prediction, prefetching, speculative execution
- Instruction-level parallelism
- Deeper memory hierarchy

Serial computation

Software traditionally written for serial computation

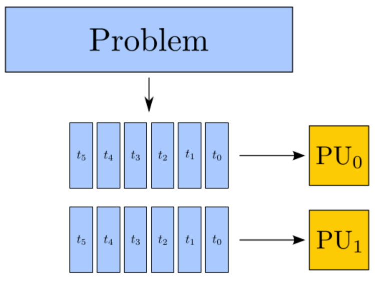
- the sequence of instructions that forms the problem is executed by one Processing Unit (PU)
- every instruction has to wait for the previous one to be completed before its execution can start
- at any moment in time, only one instruction may execute


Problem
t_{10} t_9 t_8 t_7 t_6 t_5 t_4 t_3 t_2 t_1 t_0
PU

You would like to move from a sequence stream of instructions to this new situation:

Parallel computation

In parallel computation, if two instructions have no data dependency, they can be executed in parallel, at the same time, by two PUs


Problem
t_5 t_4 t_3 t_2 t_1 t_0
PU_0
t_5 t_4 t_3 t_2 t_1 t_0
PU_1

How much can you push the parallelization?

Amdahl's Law

The maximum theoretical throughput is limited by Amdahl's Law:

- Every program contains a serial part
- Only one PU can execute the serial part
- The speedup using p PUs is given by

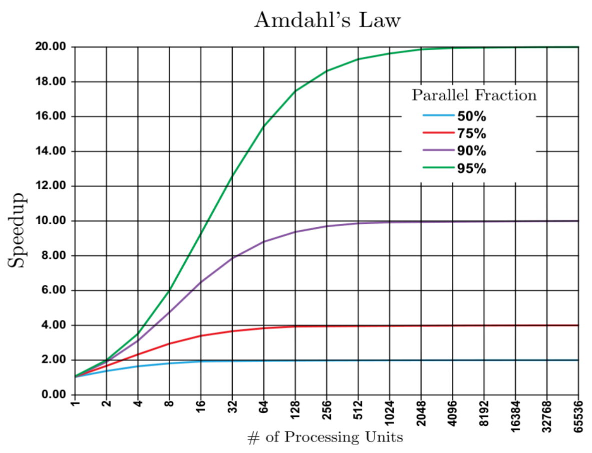
$$S(p) = \frac{T_s}{T_p}$$

- If f is the fraction of the program that runs serially, the parallel execution time is given by

$$T_p = fT_s + \frac{(1-f)T_s}{p}$$

The speed-up becomes

$$S(p,f) = \frac{T_s}{fT_s + \frac{(1-f)T_s}{p}} \rightarrow \frac{1}{f} = S_{\max}(f)$$



Mitigating Amdahl's Law: Gustafson's Law

- Amdahl's Law assumes that a problem can be split into a number of independent chunks n that can be processed in parallel and that this number is fixed.
- Many times, the increase in the size of a problem does not correspond to a growth of the sequential part:
    - Increasing the size of the problem does not change the time spent executing the sequential part, and only affects the parallel portion.
- Let f(n) be the sequential code fraction of the program:

$$S(n) = f(n) + p[1 - f(n)]$$

- f(n) decreases to 0 as n approaches infinity.
- The maximum speedup is then given by:

$$S_{\max} \equiv \lim_{n\to\infty} S(n) = p$$

It's still worth learning parallel computing: computations involving arbitrarily large data sets can be efficiently parallelized!

Parallel Computing

embarrassingly parallel problems

Problems where a large problem can be divided into many independent tasks that can be executed in parallel without needing to communicate with each other. Each task y_i is computed as a function $f_i(x_i)$, with x_i being its independent input.

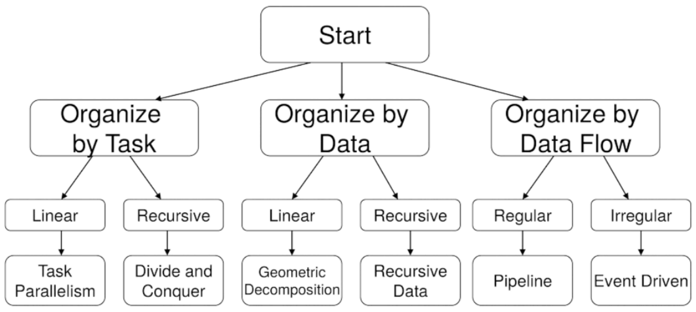
1. **Independence of Subtasks:** Each computation $y_i = f_i(x_i)$ is completely independent of the others, making parallelization straightforward. There's no data sharing or dependency between $x_0, x_1, \ldots, x_8$
2. **High Parallelism:** Tasks can be distributed across multiple processors or nodes, achieving maximum utilization of computational resources.
3. **Examples:**
    - **Linear Algebra:** Matrix-vector multiplication or diagonal element computations.
    - **Image Processing:** Applying filters or transformations pixel-by-pixel.
    - **Monte Carlo Simulation:** Running independent simulations to estimate probabilities.
    - **Cryptomining:** Hash computations for each candidate solution.
    - **Weather Forecasting:** Modeling distinct geographic regions independently.
    - **Software Compilation:** Compiling files in a project independently if there are no dependencies.

Terminology

Here some terminology

- Granularity: size of tasks
- Scheduling: order of assignment of tasks
- Mapping: assignment of tasks to a PU
- Load balancing: the art of making the computation of multiple tasks end at the same time
- Barrier: a checkpoint at which all the parallel workers should wait for the last one
- Speedup: ratio between the time of the serial application and the time of the parallel application
- Efficiency: ratio of the speedup and the number of PUs
- Race condition: When the result of execution depends on sequence and/or timing of events. Result could be incorrect if this is not taken in consideration
- Critical section: Piece of code that only one worker at a time can execute
- .
- .

Patterns for Parallel Programming



This chart is like a guide for organizing a parallel programming problem. Imagine you're trying to solve a big problem by breaking it into smaller pieces. There are three main ways to do this:

1. **By Task:** You divide the problem into different actions or tasks. For example, if you have a list of things to do, you can either work on them in order (linear) or break the tasks into smaller ones and solve those (recursive).
2. **By Data:** Instead of tasks, you split the problem based on the data itself. For instance, if you have a big spreadsheet, you might cut it into rows or sections (linear) or divide it into parts that depend on each other (recursive).
3. **By Data Flow:** This is about how the information moves between the parts. Sometimes, the flow is predictable like an assembly line (regular). Other times, it's chaotic and depends on random events (irregular).

Example: reduction

A reduction is a very common pattern in parallel computing:

- Large input data structure distributed across many PU
- Every PU independently processes and computes results for the portion of the data assigned to it (tally).
- These tally values are combined to produce the final result

Examples:

- The sum of the elements of an array
- The maximum/minimum element of an array
- Find the first occurrence of x in an array

Not parallelizeable algorithm: `accumulate`

Count number of 5s

Those are 2 different ways to count numbers

```
array[N]
numberOf5 = 0
for i in [0,N[:
  if array[i] == 5:
    ++numberOf5
```

```
numberOf5 = 0
nWorkers = 4
count5(array, workerId):
  beg = workerId*N/nWorkers
  end = beg + N/nWorkers
  for i in [beg,end[:
    if array[i] == 5:
      ++numberOf5
```

This is pseudo code, in C++ you have to use **Threads**.

Threads

A thread is an execution context, a set of register values.

- It defines the instructions to be executed and their order

A CPU core fetches this execution context and starts running the instructions: the thread is running

When the CPU needs to execute another thread, it switches the context , i.e. it saves the previous context and loads the new one

- Context switching is expensive
- Especially if threads jump from a CPU core to another

Threads enable concurrency

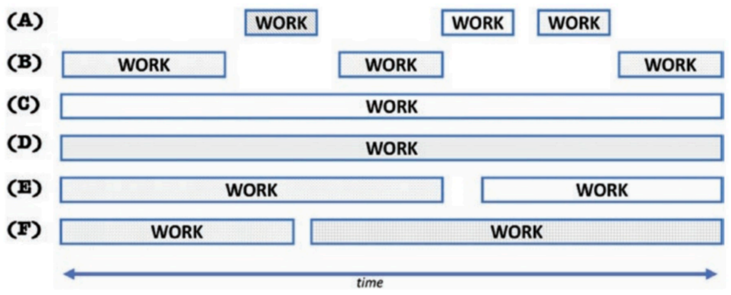
Concurrency does not imply parallelism

- If your program contains independent parts, they are the perfect candidates for running concurrently

Restaurant for dinner: - cooking food and preparing the tables are independent tasks and they can be performed by different workers to gain a speed-up

A and B are concurrent but not parallel wrt to each

- Considered all together, they are parallel



std::threads - Hello World

```
#include <thread>

int main() {

        auto f = [](int i) {
                std::cout << "hello world from thread " << i << '\n';
         };

        std::thread t0(f,0);
        std::thread t1(f,1);
        std::thread t2(f,2);

        t0.join();
        t1.join();
        t2.join();
}
```

Fork-join

- The construction of a thread is asynchronous, fork
- Threads execute independently
- A join is the synchronization point with the main thread

Measuring time

```
#include <chrono>
...

auto start = std::chrono::steady_clock::now();

f(i);

auto stop = std::chrono::steady_clock::now();

std::chrono::duration dur = stop - start;

std::cout << dur.count() << " seconds\n";
```

f() is the function that you want to measure.

Be careful, asynchronous functions return immediately: remember to synchronize before stopping the timer.

Exercise 1

You want to sum the elements of a vector in parallel using 4 threads.

- Accumulate the sum in the variable sum

```
#include <iostream>
#include <vector>
#include <thread>

void partialSum(const std::vector<int>& vec, int start, int end, int& result) {
    result = 0;
    for (int i = start; i < end; ++i) {
        result += vec[i];
    }
}

int main() {
    // Create a vector with 100 elements, all set to 1
    std::vector<int> vec(100, 1);

    int numThreads = 4;
    int chunkSize = vec.size() / numThreads;

    int partialResults[4] = {0};
    std::thread threads[4];

    // Start threads
    for (int i = 0; i < numThreads; ++i) {
        int start = i * chunkSize;
        int end = start + chunkSize;
        if (i == numThreads - 1) {
            end = vec.size();
        }
        threads[i] = std::thread(partialSum, std::ref(vec), start, end,
std::ref(partialResults[i]));
    }

    // Wait for threads to finish
    for (int i = 0; i < numThreads; ++i) {
        threads[i].join();
    }

    // Combine results
    int sum = 0;
    for (int i = 0; i < numThreads; ++i) {
        sum += partialResults[i];
    }

    std::cout << "Sum of vector elements: " << sum << std::endl;

    return 0;
}
```

.

.

Data Race

- A race condition occurs when multiple tasks read from and write to the same memory without proper synchronization.
- The “race” may finish correctly sometimes and therefore complete without errors, and at other times it may finish incorrectly.
- If a data race occurs, the behavior of the program is undefined.

`std::mutex`

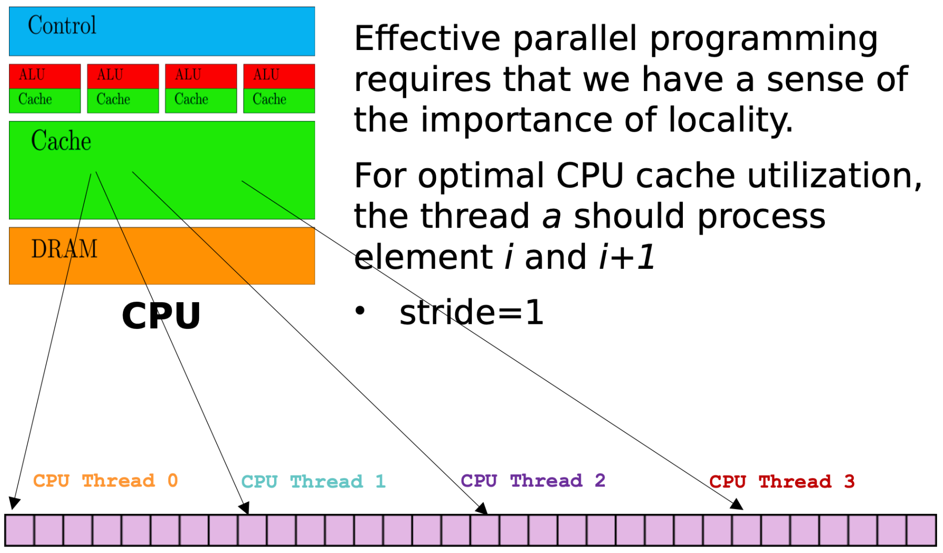
A `std::mutex` (short for "mutual exclusion") is a synchronization primitive used to protect shared data from being accessed by multiple threads simultaneously. It helps to avoid race conditions.

Scoped Lock with `std::lock_guard`

- A `std::lock_guard` is a RAII (Resource Acquisition Is Initialization) wrapper around a mutex. It locks the mutex when the `std::lock_guard` is created and automatically unlocks it when the `std::lock_guard` goes out of scope.
- This ensures that the mutex is properly unlocked, even if an exception is thrown, making the code safer.

```
#include<mutex>
std::mutex myMutex;
...
{
        std::lock_guard myLock(myMutex);
        //critical section begins here
        std::cout << "Only one thread at a time" << std::endl;

} // ends at the end of the scope of myLock
```

Memory access patterns: cached



Effective parallel programming requires that we have a sense of the importance of locality.

For optimal CPU cache utilization, the thread *a* should process element *i* and *i+1*

- stride=1

False Sharing

It's a performance issue in multithreading where threads inadvertently share the same cache line but access different memory locations. Here's a breakdown of the explanation:

Cache Line Basics:

- A **cache line** is a fixed-sized block of memory used as the smallest unit of information exchange between processor caches.
- Common cache line size is **64 bytes**.

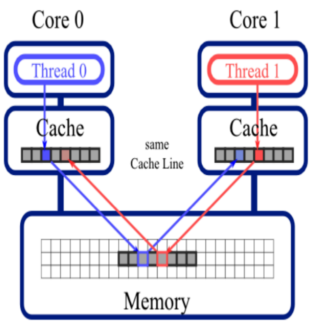
False sharing occurs when **two threads access different memory locations** that reside on the **same cache line**.

- Even though threads are working on different memory, modifications by one thread force the cache line to be reloaded in the other thread's processor.

Performance Impact:
- Cache lines have to be moved between processor caches, which may take **hundreds of clock cycles**.
- Leads to **cache invalidation** and frequent reloading of the same cache line.

Example Scenario:

- Two threads (`x` and `y`) run on two cores sharing the same cache.
- Assume:
  - `A[500]` and `B[500]` are arrays.
  - The end of `A` and the start of `B` occupy the **same cache line**.
- Thread `x` modifies `A[499]` and loads the cache line into its core.
- Thread `y` modifies `B[0]`.
  - This causes the **cache line** to be flushed and reloaded in both cores.
- Result: **Performance degradation** due to unnecessary cache line contention.



Solution:

- **Avoid false sharing** by aligning data structures to cache line boundaries:

```
#include <new>

struct alignas(std::hardware_destructive_interference_size) alignedInt {
    int x;
};
```

- `alignas(std::hardware_destructive_interference_size)` ensures that each variable is aligned to a **cache-line size**, avoiding conflicts.

std::atomic

Atomic types:

- encapsulate a value whose access is guaranteed to not cause data races
- other threads will see the state of the system before the operation started or after it finished, but cannot see any intermediate state
- can be used to synchronize memory accesses among different threads

```
#include
std::atomic<int> x = 0;
int a = x.fetch_add(42);
```

- reads from a shared variable, adds 42 to it, and writes the result back: all in one indivisible step

Trivially Copyable

A type is **trivially copyable** if:

- It can be copied by **bit-wise memory copying** (e.g., using `memcpy` to copy its bytes directly).
- It does not need any special logic to copy its values, like constructors, destructors, or virtual functions.

Characteristics:

- **Continuous memory:** The object is stored in a single, uninterrupted block of memory, making it efficient to copy.
- **Copying = bit-by-bit clone:** When copied, all its bits are duplicated exactly, without calling any functions or involving the object's internal logic.
- **No complex features:** The type can't have virtual functions (like those in polymorphic objects), and its constructor must not throw exceptions.

Example:

- Types like `int`, `double`, or structs made up of these basic types are trivially copyable:

  ```
  std::atomic<int> i;       // OK: int is trivially copyable
  std::atomic<double> x;    // OK: double is trivially copyable
  struct S { long x; long y; }; // A struct with basic types
  std::atomic<S> s;         // OK: Struct S is trivially copyable
  ```

- Why does this matter for `std::atomic`? Atomic operations often rely on low-level memory operations like bit-level copying or exchanging, so they require the type to be simple.

std::atomic

Without atomicity, two threads might read the same value simultaneously, leading to incorrect results. With `std::atomic`, these operations are handled safely.

Key Features:

1. **Atomicity for Safety:**
   - Operations like `++x` (increment) or `x = y` (assignment) happen as a single step. This guarantees that no other thread can interrupt or access the variable during the operation.

2. **Operator Overloads:**
    - `std::atomic` provides operator overloads for common actions (like `+=` or `|=`), but only when these operations can be safely performed atomically.

Examples:

1. **Atomic Initialization:**

    ```
    std::atomic<int> x{0}; // Initialize an atomic integer to 0
    ```

    Here, `x` is now thread-safe for operations like incrementing or assigning.
2. **Atomic Increment and Decrement:**
    - `++x` and `x++` are atomic because `std::atomic` ensures that only one thread can access and modify the value at a time.
3. **Non-atomic Operations:**
    - Operations that involve multiple steps, like `x *= 2`, are **not atomic** and will not compile because they cannot guarantee thread safety.
4. **Atomic Read/Write:**

    ```
    int y = x * 2;   // Atomic read of x
    x = y + 1;       // Atomic write to x
    ```

    When reading or writing to `x`, it is guaranteed that no other thread is modifying it at the same time.
5. **Atomic Exchange:**
    - `std::atomic` provides the `exchange()` method to safely swap values: `cpp int z = x.exchange(y); // Atomically: z = x; x = y;` This ensures that the swap operation is done in one step, with no interruptions.

Why is this useful?

In multithreaded environments, shared variables are often accessed by multiple threads. Without atomicity, race conditions can occur, leading to unpredictable bugs. `std::atomic` solves this by making these operations thread-safe.

Here's an explanation of the topics in the image, enhanced with details and practical insights:

Compare-and-Swap (CAS)

CAS is a **low-level atomic operation** that is commonly used to implement lock-free data structures. It works by comparing the current value of a variable with an expected value and, if they match, updating the variable to a new value.

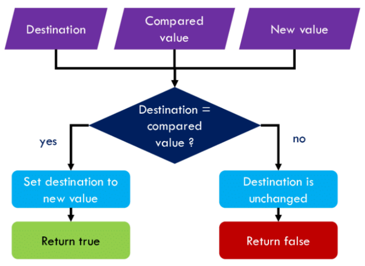
```
bool success = x.compare_exchange_weak(y, z);
```

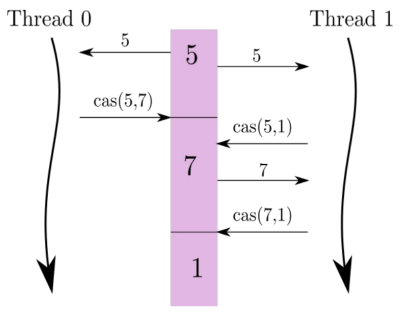
- **x:** The atomic variable.
- **y:** The expected value.

- **z:** The new value.

If the current value of `x` matches `y`:

- The value is updated to `z`, and `success` is set to `true`. Otherwise:
- The operation fails, `y` is updated with the current value of `x`, and `success` is set to `false`.





- Two threads (Thread 0 and Thread 1) attempt to update the value of a variable using CAS.
- Example:
  - Initial value: 5
  - Thread 0: Executes `cas(5,7)` (compare 5; if 5, replace with 7) → Success, value becomes 7.
  - Thread 1: Executes `cas(5,1)` → Fails because the value is now 7.

Key Advantages:

- **Lock-Free:** CAS allows for safe concurrent modifications without the need for traditional locks (like `mutex`), reducing contention.
- **Performance:** Useful in performance-critical systems where locking mechanisms can introduce latency.

Data Hazards

What are Data Hazards?

- Data hazards occur when multiple threads access shared memory without proper synchronization, leading to unpredictable behavior.
- **Pro:** Threads can communicate using shared memory.
- **Con:** Without synchronization, this communication can lead to errors or race conditions.

Types of Data Hazards:

1. **Read-After-Write (RAW):**
   - Thread reads a value before another thread finishes writing to it.
   - Example: A thread reads a variable `x` as 5 while another thread is incrementing it to 6.
2. **Write-After-Read (WAR):**
   - A thread writes to a variable after another thread reads it, potentially invalidating the read value.

    - Example: A thread reads `x` as 5, then another thread immediately writes 10 to `x`.
3. **Write-After-Write (WAW):**
    - Two threads write to the same variable at the same time, leading to loss of one write.
    - Example: Thread 1 writes 6 to `x`, but Thread 2 overwrites it with 7 simultaneously.

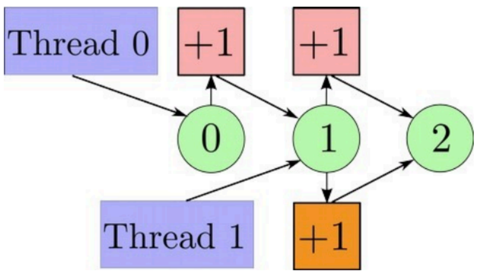
Implications:

- **Race Conditions:** Data hazards result in race conditions, where the final value depends on the thread execution order, making it unpredictable and hard to debug.

Avoiding Data Hazards

Solutions:

- **Thread-Safe Code:**
    - A program is thread-safe when it avoids data hazards and ensures that shared memory access is synchronized.
- **Critical Sections:**
    - Use synchronization mechanisms like a `mutex` (mutual exclusion) to ensure that only one thread can access a shared resource at a time.



Example (Using Mutex):

```
lock();
shared_variable++;
unlock();
```

- The `lock()` ensures no other thread can modify `shared_variable` while the current thread is updating it.

Example: Counting the Number of 5s in an Array

Problem:

- Given an array, count how many elements are equal to 5 using multiple threads.

Approach:

- Divide the array into equal parts, with each thread responsible for a portion.

- Use a shared variable (`numberOf5`) to count the 5s.
- Synchronize access to `numberOf5` using a `lock` to prevent race conditions.

Pseudocode:

```
int array[N];
int numberOf5 = 0;
int nWorkers = 4;

void count5(int workerId) {
    int beg = workerId * N / nWorkers;
    int end = beg + N / nWorkers;
    for (int i = beg; i < end; i++) {
        if (array[i] == 5) {
            lock();
            ++numberOf5;
            unlock();
        }
    }
}
```

Explanation:

- The array is divided into 4 parts, and each thread processes its part independently.
- Whenever a thread finds a 5, it locks access to the shared `numberOf5` variable, increments it, and unlocks.

Contention

- When multiple threads (or processing units) try to update the same data at the same time, the system is forced to serialize those updates to avoid data corruption.
- This can lead to a performance bottleneck.

  Massively Parallel Execution Cannot Afford Serialization
- In parallel programming, you want to divide the work among many threads so they can run simultaneously.
- If a critical piece of data must be updated often (e.g., a shared counter), each update might be forced into a "one-at-a-time" mode—ruining the benefits of parallelism.

The more threads that try to access or update a single shared resource, the greater the contention—and thus the overhead.

Key takeaway: If every thread must frequently update the same variable (e.g., `numberOf5`), these updates act as a choke point.

Mitigating Contention

1. **Privatization**
   - Give each thread or worker its own copy of the data to reduce contention on a global structure.

    - Each thread works on its local/private version; updates to the global data only happen once per thread or in bulk.
2. **Transformation of the Access Pattern**
    - Restructure how you access data so there are fewer shared accesses.
    - For instance, instead of each thread incrementing the same counter repeatedly, each thread can accumulate a private total and then do a single update to a global counter.
3. **Avoid Frequent Transactions to/from Global Memory**
    - Memory transactions (especially to global/main memory in a GPU or multi-core system) are expensive.
    - Minimize how often you read/write to global data.
4. **Use Registers and Shared Memory for Aggregating Partial Results**
    - These faster memory tiers reduce overhead.
    - Summarizing partial results locally (e.g., in a register or thread-local variable) is quicker than multiple global updates.
5. **Requires Storage Resources to Keep Copies of Data**
    - The downside of privatization is the extra memory cost (each thread has its own copy).

Example Code: Counting the Number of 5s

- **Serial version**

```
array[N]
numberOf5 = 0
for i in [0, N[:
    if array[i] == 5:
        numberOf5++
return numberOf5
```

    - Simple loop, single thread.
- **Parallel version (with 4 workers)**

```
numberOf5 = 0
nWorkers = 4

count5(array, workerId):
    privateResult = 0
    beg = workerId * (N/nWorkers)
    end = beg + N/nWorkers

    for i in [beg, end[:
        if array[i] == 5:
            privateResult++

    lock()
    numberOf5 += privateResult
    unlock()
```

1. Each thread/worker gets a chunk of the array (determined by `beg` and `end`).

2. A private counter (`privateResult`) is maintained per thread.
3. After counting within its chunk, each thread does a single update to the shared `numberOf5` (protected by lock/unlock).

Key takeaway: This approach greatly reduces contention compared to incrementing a global counter for every single `5` found. Instead, each thread only updates the global counter once.

---

4. Performance Chart and the Question

> **"The T=8 version does not take half the time compared to T=4. Why not?"**

Reasons for Non-Linear Speedup

Increasing the number of threads to 8 does not simply halve the time from 4 threads because real-world parallel programs face overheads from synchronization, memory bottlenecks, non-ideal load balancing, and diminishing returns as you keep adding threads.

Load Balancing

Sometimes dividing the input data in 2 does not mean that the load has been also divided.

- Example: total load 100. If 5 workers take 20 each we have a speedup of 5, if 1 worker takes 50, we have speedup of 2.
- Non-uniform data distributions
- Highly concentrated spatial data areas
- Astronomy, medical imaging, computer vision, rendering

If each thread processes the input data of a given spatial volume unit, some will do a lot more work than others.

Load Imbalance with Code Example

```
i_start = my_id * (N / num_threads);  // Start index for this thread
i_end = i_start + (N / num_threads); // End index for this thread
if (my_id == (num_threads - 1)) i_end = N; // Adjust for the last thread

for (int i = i_start; i < i_end; ++i) {
    ...
}
```

1. **How Work is Divided:**
   - N/num_threadsN / num_threads: Each thread gets an equal share of iterations.
   - `i_start` and `i_end`: Define the range for each thread based on its ID (`my_id`).
   - The **last thread** (`my_id == num_threads - 1`) adjusts `i_end` to ensure all remaining iterations are included.

2. **With N = 1000 and `num_threads=32`:**
    - Each thread is assigned $\lfloor N/num\_threads \rfloor = 31 iterations.$
    - The first 31 threads execute iterations: $0 \text{ to } 30,\ 31 \text{ to } 61, \ldots, 961 \text{ to } 991$
    - The **last thread** adjusts `i_end` to NN, handling the leftover: $992 \text{ to } 999$
3. **Why Load Imbalance Occurs:**
    - The last thread processes $31 + 8 = 39$ iterations, while others only process 31.
    - This happens because N is not perfectly divisible by $num\_threads$

Partitioning and Load Balancing

- Parallel computing All exponential laws come to an end...

Parallel computing becomes useful when:

- The solution to our problem takes too much time (but consider Amdahl's Law)
- The size of our problem is big (Gustafson's Law)
- The solution of our problems is poor, we would like to have a better one
  Three steps to a better parallel software:
    1. Restructure the mathematical formulation
    2. Innovate at the algorithm and data structure level
    3. Tune core software for the specific architecture

Threading Building Blocks

OneAPI Threading Building Blocks (TBB) is a library proposed by Intel which allows to express parallelism on CPUs in a C++ program.

- Parallelizing for loops can be tedious with `std::thread`
- One wants to achieve scalable parallelism, easily
  To use the TBB library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner
- We will see just the `parallel_for` construct

Why not threads directly?

Direct programming with threads forces you to do the work to efficiently map (schedule) logical tasks onto threads

- The TBB runtime library maps tasks onto threads to maximize load balancing and, hence, performance

Example:

```
for (int i = 0; i != N; ++i) {
        ++x[i];
}
```

```
oneapi::tbb::parallel_for(
        oneapi::tbb::blocked_range<int>(0, N),
        [&](oneapi::tbb::blocked_range<int> range) {
                for (int i = range.begin(); i != range.end(); ++i) {
                        ++x[i];
                }
        }
);
```

Scalability

A loop needs to last for at least 1M clock cycles for parallel_for to become worth it.

Usually, adding more cores than the limit does not only result in performance improvements, but performance falls.

- Overhead in scheduling and synchronizing many small tasks starts dominating

TBB uses the concept of Grain Size to control the granularity of tasks

Grain Size

The grain size affects how the scheduler can distribute tasks to the available workers (threads)

- If the grain size is 1000 and the loop iterates over 2000 elements, the scheduler can give the work at most to 2 threads

Note:

- with a grain size of 1, most of the time may be spent in scheduling
- with a grain size of N, the execution is in fact sequential

Partitioners

Automatic: TBB provides a default set of heuristics to select a good-enough grain size.

- Simple: the simple_partitioner allows to manually select the grain size, passing it to the blocked_range constructor
  - The default is 1, in units of loop iterations per chunk
  - Rule of thumb: G iterations should take at least 100k clock cycles
- Affinity: the affinity_partitioner can help when:
  - data in a loop fits in cache
  - the ratio between computations and memory accesses is low