



PROGRAMMING IN C++

Jülich Supercomputing Centre

13 – 17 May 2024 | Sandipan Mohanty | Forschungszentrum Jülich, Germany

Chapter 2

C++ classes and class hierarchies

C++ classes

OBJECTS, PROPERTIES, CLASSES

Atomic Number	Common Name	Symbol	Molar Mass (g/mol)	Valence Electrons
1	Hydrogen	H	1.008	1
2	Helium	He	4.0026	2
3	Lithium	Li	6.94	1
4	Beryllium	Be	9.0122	2
5	Boron	B	10.81	3
6	Carbon	C	12.011	4
7	Nitrogen	N	14.007	5
8	Oxygen	O	15.999	6
9	Fluorine	F	18.998	7
10	Neon	Ne	20.180	8
11	Sodium	Na	22.990	1
12	Magnesium	Mg	24.305	2
13	Aluminum	Al	26.982	3
14	Silicon	Si	28.085	4

OBJECTS, PROPERTIES, CLASSES

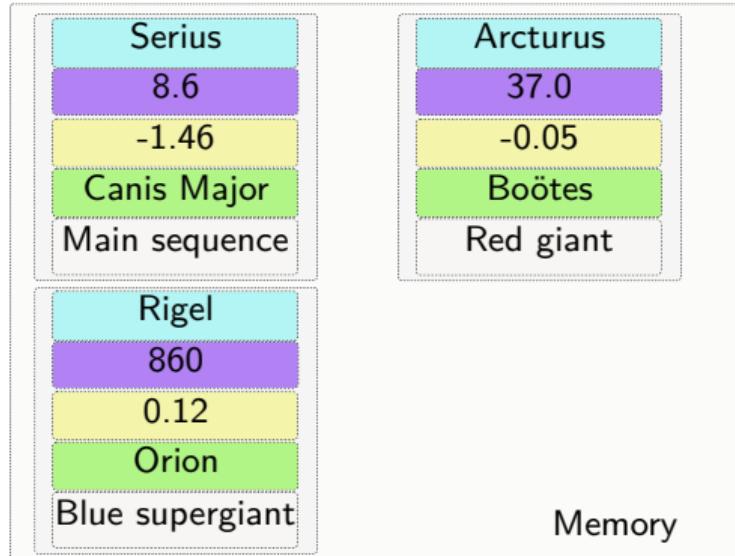
Name	Distance from Sun (light years)	Apparent Brightness	Constellation	Stellar Category
Sirius	8.6	-1.46	Canis Major	Main Sequence
Canopus	310	-0.72	Carina	Bright Giant
Arcturus	37	-0.05	Boötes	Red Giant
Alpha Centauri	4.37	-0.27	Centaurus	Binary System (Main Sequence)
Vega	25	0.03	Lyra	Main Sequence
Rigel	860	0.12	Orion	Blue Supergiant
Betelgeuse	640	0.42	Orion	Red Supergiant
Procyon	11.4	0.38	Canis Minor	Main Sequence
Achernar	139	0.46	Eridanus	Blue Subgiant
Altair	17	0.77	Aquila	Main Sequence

- Each star in the table has a certain number of properties
- Notation: `a_star.name`, `a_star.constellation` ...

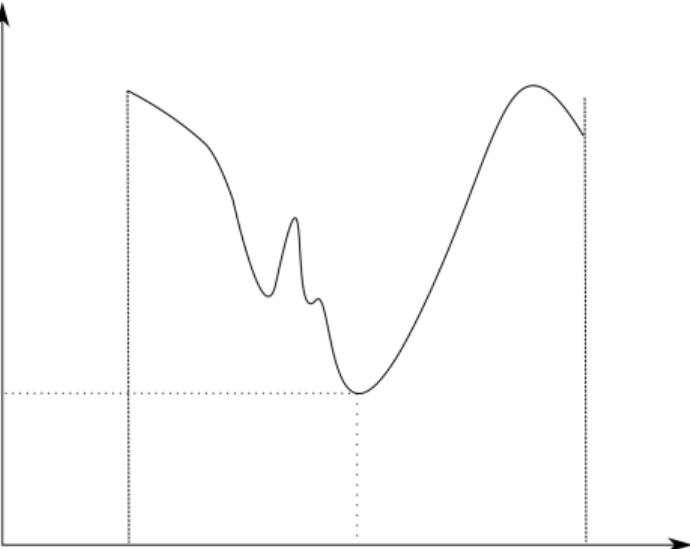
OBJECTS, PROPERTIES, CLASSES

```
1 struct Star {  
2     string name;  
3     Distance d;  
4     double brightness;  
5     string constellation, category;  
6 };
```

- Objects of a “type”
- Each “instance” in memory represents an entity with the same set of “members”, e.g., a name, a constellation etc.
- The data contained in the members varies as needed



AD HOC STRUCTS



- Some times calculations involve bundles of entities which belong together, e.g., the location of a minimum of a function and the corresponding minimum value

AD HOC STRUCTS

```
1 struct minim_return_type {
2     double min_loc, min_val;
3 };
4 auto minimize(double r1, double r2,
5               FunctionType f)
6 {
7     minim_return_type m;
8     // Find minimum somehow
9     m.min_loc = the_location;
10    m.min_val = the_value;
11    return m;
12 }
13 void elsewhere()
14 {
15     auto sol = minimize(0., 10., cost_func);
16     cout << "Minimum found at " << sol.min_loc
17         << "with a value " << sol.min_val
18         << "\n";
19 }
```

- **struct** : Staple together objects of arbitrary types
- Can be done in global as well as block scope

AD HOC STRUCTS

```
1 struct minim_ret_type {
2     double min_loc, min_val;
3 };
4 auto minimize(double r1, double r2,
5               FunctionType f)
6 {
7     minim_ret_type m;
8     // Find minimum somehow
9     m.min_loc = the_location;
10    m.min_val = the_value;
11    return m;
12 }
13 void elsewhere()
14 {
15     auto [loc, val] = minimize(0., 10.,
16                               cost_func);
17     cout << "Minimum found at " << loc
18     << "with a value " << val << "\n";
19 }
```

- **struct** : Staple together objects of arbitrary types
- Can be done in global as well as block scope
- We can now use the name of the **struct** to create variables , such that each of them has a **min_loc member** and a **min_val member**
- Can be function argument (and hence can participate in overload resolution), or return value (and hence gives us a way to return multiple values)
- Names of the components can be chosen to reflect any meanings associated with the content, making it a good practical way of returning multiple objects from a function

AD HOC STRUCTS

```
1 struct minim_ret_type {
2     double min_loc, min_val;
3 };
4 auto minimize(double r1, double r2,
5               FunctionType f)
6 {
7     minim_ret_type m;
8     // Find minimum somehow
9     m.min_loc = the_location;
10    m.min_val = the_value;
11    return m;
12 }
13 void elsewhere()
14 {
15     auto [loc, val] = minimize(0., 10.,
16                               cost_func);
17     cout << "Minimum found at " << loc
18     << "with a value " << val << "\n";
19 }
```

- **struct** : Staple together objects of arbitrary types
- Can be done in global as well as block scope
- We can now use the name of the **struct** to create variables , such that each of them has a **min_loc member** and a **min_val member**
- Can be function argument (and hence can participate in overload resolution), or return value (and hence gives us a way to return multiple values)
- Names of the components can be chosen to reflect any meanings associated with the content, making it a good practical way of returning multiple objects from a function
- Structured bindings can be used to create aliases for the components. The binding names are independent of the names in the **struct**

AD HOC STRUCTS

```
1 struct minim_ret_type {
2     double min_loc, min_val;
3 };
4 auto minimize(double r1, double r2,
5               FunctionType f)
6 {
7     minim_ret_type m;
8     // Find minimum somehow
9     m.min_loc = the_location;
10    m.min_val = the_value;
11    return m;
12 }
13 void elsewhere()
14 {
15     auto m1 = minimize(-10., 0., constfunc1);
16     minim_ret_type m2 = minimize(-10., 0.,
17                                   constfunc1);
18     auto *mptr = &m2;
19     if (m1.min_val > mptr->min_val)
20         haha();
21 }
```

- A **struct** is a user defined data type
- Each *instance* has a bundle, with a `min_loc` and `min_val` member
- Members are accessed from the object using the `.` notation, and from a pointer to an object using the `->` notation.
`(*mptr).min_val` is the same as
`mptr->min_val`



DESIGNATED INITIALISERS

```
1 // examples/desig2.cc
2 struct v3 { double x, y, z; };
3 struct pars { int offset; v3 velocity; };
4 auto operator<<(std::ostream& os, const v3& v) -> std::ostream&
5 {
6     return os << v.x << ", " << v.y << ", " << v.z << " ";
7 }
8 auto example_func(pars p)
9 {
10    std::cout << p.offset << " with velocity " << p.velocity << "\n";
11 }
12 auto main() -> int
13 {
14     example_func( {.offset = 5, .velocity = {.x=1., .y = 2., .z=3.}} );
15 }
```

-
- Simple struct type objects can be initialised by **designated initialisers** for each field.
 - Can be used to implement a kind of "keyword arguments" for functions. But remember, at least as of C++20, the field order can not be shuffled.

C++ CLASSES

```
1 // examples/trivialclassoverload.cc
2 class A {};
3 class B {};
4 void func(int i, A a)
5 {
6     cout << "Called f input types (int, A)\n";
7 }
8 void func(int i, B b)
9 {
10    cout << "Called f input types (int, B)\n";
11 }
12 auto main() -> int
13 {
14     A xa;
15     B xb;
16     func(0, xa) ;
17     func(0, xb) ;
18 }
```

- User defined data types. Independently created classes are different, even if they have the same content.
- Function overloading: The two versions of the function `func` shown here are different entities from the compiler's viewpoint. No ambiguity about which function is called in lines 16 and 17 in `main()`.

C++ CLASSES

Overloading operators

```
1 // examples/op_overload.cc
2 class A {};
3 class B {};
4 auto operator+(A x, A y) -> A
5 {
6     std::cout << "operator+(A, A)\n";
7     return x;
8 }
9 auto operator+(B x, B y) -> B
10 {
11     std::cout << "operator+(B, B)\n";
12     return x;
13 }
14 auto operator+(A x, B y) -> A {...} // similar
15 auto main() -> int {
16     A a1, a2;
17     B b1, b2;
18     a1 + a2;
19     a1 + b1;
20     b1 + b2; // b1 + a2; doesn't work. Think why!
21 }
```

- For C++ class types, operators like `+`, `-`, `*`, `/`, `||`, `&&` ... are functions
- As long as at least one of the arguments to an operator is of a class type (not a built-in type like `int`, `double` ...), it is possible to provide a recipe to interpret expressions like `a1 + a2`
- `a1 + a2` is interpreted as a function call `operator+(a1, a2)`
- Using suitably chosen operators to overload, we can make expressions involving objects of a class type more intuitive

OVERLOADING OPERATORS

+	-	*	/	%	&	^			
+=	-=	*=	/=	%=	&=	^=	=	=	=
++	--	&&		!	!=	==			
<	>	!=	==	<=	>=	<=>	=	=	=
()	[]	,	->	->*	<<	<<=	>>=	>>	
new	delete	new[]	delete[]						

Table: List of operators you can overload. (But remember, *can* and *should* are not the same thing!)

- Think carefully about the impact an overloaded operator will have on the readability of your code. Whether or not it the impact is beneficial depends on the use case
- Many important commonly used C++ features depend on suitably overloaded operators. E.g.,
`std::cout << "Hello\n";`

C++ CLASSES

```
1 struct Vector3 {  
2     double x, y, z;  
3 };
```

- Usually, encapsulates some data to represent an idea

C++ CLASSES

```
1 struct Vector3 {  
2     double x, y, z;  
3     auto mag2() -> double  
4     {  
5         return x * x + y * y + z * z;  
6     }  
7 };
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data

C++ CLASSES

```
1 struct Vector3 {
2     double x, y, z;
3     auto mag2() -> double
4     {
5         return x * x + y * y + z * z;
6     }
7 }
8
9 void somefunc()
10 {
11     int a, b, c;
12     Vector3 d, e, f;
13     // ...
14     if (d.mag2() < e.mag2()) doX();
15 }
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data
- Once defined, one can create and use variables of the new type

C++ CLASSES

```
1 struct Vector3 {
2     double x, y, z;
3     auto mag2() -> double
4     {
5         return x * x + y * y + z * z;
6     }
7 };
8
9 void somefunc()
10 {
11     int a, b, c; // On the stack
12     Vector3 d, e, f; // On the stack
13     // ...
14     if (d.mag2() < e.mag2()) doX();
15 }
```

- Usually, encapsulates some data to represent an idea
- Specifies possible operations on the data
- Once defined, one can create and use variables of the new type

In C++, objects of user defined types live on the stack by default, unless explicitly created on the heap.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the **struct** :

- Data and function members

```
1 struct complex {
2     double real, imaginary;
3     auto modulus() -> double
4     {
5         return sqrt(real * real +
6                     imaginary * imaginary);
7     }
8 };
9 ...
10 complex a{1, 2}, b{3, 4};
11 complex* cptr = &a;
12 auto c = a.modulus(); // 1 * 1 + 2 * 2
13 auto d = b.modulus(); // 3 * 3 + 4 * 4
14 auto e = cptr->modulus(); // 1 * 1 + 2 * 2
```

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
1 struct complex {
2     double real, imaginary;
3     auto modulus() -> double
4     {
5         return sqrt(real * real +
6                     imaginary * imaginary);
7     }
8 };
9 ...
10 complex a{1, 2}, b{3, 4};
11 complex* cptr = &a;
12 auto c = a.modulus(); // 1 * 1 + 2 * 2
13 auto d = b.modulus(); // 3 * 3 + 4 * 4
14 auto e = cptr->modulus(); // 1 * 1 + 2 * 2
```

- Data and function **members**
- A (non-static) member function is invoked on an **instance** of our structure.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
1 struct complex {
2     double real, imaginary;
3     auto modulus() -> double
4     {
5         return sqrt(real * real +
6                     imaginary * imaginary);
7     }
8 };
9 ...
10 complex a{1, 2}, b{3, 4};
11 complex* cptr = &a;
12 auto c = a.modulus(); // 1 * 1 + 2 * 2
13 auto d = b.modulus(); // 3 * 3 + 4 * 4
14 auto e = cptr->modulus(); // 1 * 1 + 2 * 2
```

- Data and function **members**
- A (non-static) member function is invoked on an **instance** of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of `a`.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
1 struct complex {
2     double real, imaginary;
3     auto modulus() -> double
4     {
5         return sqrt(real * real +
6                     imaginary * imaginary);
7     }
8 };
9 ...
10 complex a{1, 2}, b{3, 4};
11 complex* cptr = &a;
12 auto c = a.modulus(); // 1 * 1 + 2 * 2
13 auto d = b.modulus(); // 3 * 3 + 4 * 4
14 auto e = cptr->modulus(); // 1 * 1 + 2 * 2
```

- Data and function **members**
- A (non-static) member function is invoked on an **instance** of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of `a`.
- Inside a member function, member variables of the invoking instance can be referred to as `this->varname`, which can be shortened to just `varname`.

C++ CLASSES

Functions, relevant for the idea, can be declared inside the `struct` :

```
1 struct complex {
2     double real, imaginary;
3     auto modulus() -> double
4     {
5         return sqrt(real * real +
6                     imaginary * imaginary);
7     }
8 };
9 ...
10 complex a{1, 2}, b{3, 4};
11 complex* cptr = &a;
12 auto c = a.modulus(); // 1 * 1 + 2 * 2
13 auto d = b.modulus(); // 3 * 3 + 4 * 4
14 auto e = cptr->modulus(); // 1 * 1 + 2 * 2
```

- Data and function `members`
- A (non-static) member function is invoked on an `instance` of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of `a`.
- Inside a member function, member variables of the invoking instance can be referred to as `this->varname`, which can be shortened to just `varname`.
- Think of a call like `a.modulus()` as `complex::modulus(a)`. The address of the object on the left of the `.` is the implicit first argument to the member function.

The screenshot shows two side-by-side compiler windows in the Compiler Explorer interface.

Left Window (C++ Source):

```
1 struct Example {  
2     double x, y;  
3     auto mod() -> double;  
4 };  
5  
6 auto Example::mod() -> double  
7 {  
8     return x * x + y * y;  
9 }  
10  
11 auto unrelated(Example* ptr) -> double  
12 {  
13     return ptr->x * ptr->x + ptr->y * ptr->y;  
14 }
```

Right Window (Assembly Output):

```
1 Example::mod():                                # @Example::mod()  
2     movupd  xmm1, xmmword ptr [rdi]  
3     mulpd  xmm1, xmm1  
4     movapd  xmm0, xmm1  
5     unpckhd  xmm0, xmm1  
6     addsd  xmm0, xmm1  
7     ret  
8 unrelated(Example*):                           # @unrelated(Example*)  
9     movupd  xmm1, xmmword ptr [rdi]  
10    mulpd  xmm1, xmm1  
11    movapd  xmm0, xmm1  
12    unpckhd  xmm0, xmm1  
13    addsd  xmm0, xmm1  
14    ret
```

The assembly output shows that the compiler has generated SIMD instructions (movupd, mulpd, movapd, unpckhd, addsd) for the multiplication and addition operations. The `unrelated` function is shown to have identical assembly code as the `mod` function, indicating that the compiler has optimized away the pointer arithmetic and instead uses the same SIMD operations for both functions.

Compiler Explorer + [New](#)

<https://www.godbolt.org/>

Arch Linux Forums Other Bookmarks

COMPILER EXPLORER Add... More Templates Share Policies Other

C++ source #1 X

A C++

```

1 struct Example {
2     double x, y;
3     auto mod() -> double;
4     auto mod23(this Example&) -> double;
5 };
6
7 auto Example::mod() -> double
8 {
9     return x * x + y * y;
10 }
11 auto unrelated(Example* ptr) -> double
12 {
13     return ptr->x * ptr -> x + ptr->y * ptr->y;
14 }
15
16 auto Example::mod23(this Example& self) -> double
17 {
18     return self.x * self.x + self.y * self.y;
19 }
20

```

x86-64 gcc (trunk) (Editor #1) X

x86-64 gcc (trunk) -O3 -std=c++23 -march=skylake

A C++

```

1 Example::mod():
2     vmovsd xmm1, QWORD PTR [rdi+8]
3     vmovsd xmm0, QWORD PTR [rdi]
4     vmulsd xmm1, xmm1, xmm1
5     vfmadd132sd    xmm0, xmm1, xmm0
6     ret
7 unrelated(Example*):
8     vmovsd xmm1, QWORD PTR [rdi+8]
9     vmovsd xmm0, QWORD PTR [rdi]
10    vmulsd xmm1, xmm1, xmm1
11    vfmadd132sd    xmm0, xmm1, xmm0
12    ret
13 Example::mod23(this Example&):
14     vmovsd xmm1, QWORD PTR [rdi+8]
15     vmovsd xmm0, QWORD PTR [rdi]
16     vmulsd xmm1, xmm1, xmm1
17     vfmadd132sd    xmm0, xmm1, xmm0
18     ret

```

C Output (0/0) x86-64 gcc (trunk) i - 541ms (7883B) ~499 lines filtered Compiler License

EXPLICIT THIS PARAMETER

- Non-static member functions of a class define a local pointer variable called `this` inside the function, which points to the invoking instance
- In C++23, the implicit invoking instance can instead be written out explicitly, similar to the `self` parameter for class member functions in Python.
- More on this after we learn about templates!

OPERATORS AS MEMBER FUNCTIONS

```
1 struct complex {
2     double real, imag;
3     auto modulus() -> double
4     {
5         return sqrt(real * real +
6                     imag * imag);
7     }
8     auto operator+(complex other) -> complex
9     {
10        return {real + other.real,
11                 imag + other.imag};
12    }
13};
```

- Since operators working with class types are normal functions, one can have operators as member functions
- The implicit argument (invoking instance) is on the left hand side for binary operators. That's why the binary operator `+` is defined here as a member function taking only one argument

MEMBER FUNCTIONS AND CONST

```
1 struct complex {
2     double m_real, m_imag;
3     auto modulus() -> double;
4     auto operator-( const complex& rhs)
5             -> complex;
6 };
7 auto operator*( const complex* lhsptr ,
8                     const complex& rhs )
9             -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // But did z1 ?
19 }
```

- Explicit arguments to member functions can be declared `const` similar to arguments for any other function

MEMBER FUNCTIONS AND CONST

```
1 struct complex {
2     double m_real, m_imag;
3     auto modulus() -> double;
4     auto operator-( const complex& rhs)
5         -> complex;
6 };
7 auto operator*( const complex* lhsptr ,
8                     const complex& rhs )
9     -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // But did z1 ?
19 }
```

- Explicit arguments to member functions can be declared `const` similar to arguments for any other function
- Non-member operator definitions like `operator*` in line 6 have two parameters: the first represents the left side and the second the right side. These are easy to declare `const`.

MEMBER FUNCTIONS AND CONST

```
1 struct complex {
2     double m_real, m_imag;
3     auto modulus() -> double;
4     auto operator-( const complex& rhs)
5             -> complex;
6 };
7 auto operator*( const complex* lhsptr ,
8                     const complex& rhs )
9             -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // But did z1 ?
19 }
```

- But member functions have an implicit argument: the `this` pointer, pointing at the calling instance. In member operators, `this` points at the object on the left side. Only the right side is passed explicitly in a member operator definition (line 4).
- Where do we put a `const` qualifier, if we want to express that the calling instance (LHS argument) must not change?

MEMBER FUNCTIONS AND CONST

```
1 struct complex {
2     double m_real, m_imag;
3     auto modulus() const -> double;
4     auto operator-( const complex& b) const
5         -> complex;
6 };
7 auto operator*( const complex* lhsptr,
8                     const complex& rhs )
9     -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // We know z1 didn't change,
19     // as we called a const member
20 }
```

- But member functions have an implicit argument: the `this` pointer, pointing at the calling instance. In member operators, `this` points at the object on the left side. Only the right side is passed explicitly in a member operator definition (line 4).
- Where do we put a `const` qualifier, if we want to express that the calling instance (LHS argument) must not change?
- Answer: After the closing parentheses of the function signature.

MEMBER FUNCTIONS AND CONST

```
1 struct complex {
2     double m_real, m_imag;
3     auto modulus() const -> double;
4     auto operator-( const complex& b) const
5         -> complex;
6 };
7 auto operator*( const complex* lhsptr,
8                     const complex& rhs )
9     -> complex
10 {
11     // ...
12 }
13 void somewhere_else()
14 {
15     complex z1, z2;
16     auto z3 = z1 - z2;
17     // We know z2 didn't change.
18     // We know z1 didn't change,
19     // as we called a const member
20 }
```

- But member functions have an implicit argument: the `this` pointer, pointing at the calling instance. In member operators, `this` points at the object on the left side. Only the right side is passed explicitly in a member operator definition (line 4).
- Where do we put a `const` qualifier, if we want to express that the calling instance (LHS argument) must not change?
- Answer: After the closing parentheses of the function signature.
- The `const` qualifier does not refer to the function, but to the implicit argument

SOME EXAMPLE CLASSES

```
1 class Angle {
2     double rd = 0;
3 public:
4     enum unit {
5         radian,
6         degree
7     };
8     Angle operator-(Angle a) const ;
9     Angle operator+(Angle a) const ;
10    Angle operator+=(Angle a) ;



---



```
1 class Vector3
2 {
3 public:
4 enum crdtype {cartesian=0,polar=1};
5 inline auto x() const -> double {return dx;}
6 inline void x(double gx) {dx=gx;}
7 auto dot(const Vector3 &p) const -> double;
8 Vector3 cross(const Vector3 &p) const;
```


```

```
1 class IsingLattice {
2 public:
3     using update_type =
4         std::pair<size_t, size_t>;
5     IsingLattice();
6     IsingLattice(size_t Nx, double JJ);
7     void setLatticeSize(size_t ns);



---



```
1 class KMer {
2 public:
3 Nucleotide at(size_t i);
4 auto operator==(const KMer &) const -> bool;

```
1 class SimulationManager {
2 public:
3     void loadSettings(std::string file);
4     auto checkConfig() const -> bool;
5     void start();
```


```


```

OBJECT INITIALISATION: CONSTRUCTORS

- In C++, initialisation functions for a struct have the same name as the struct. They are called *constructors*.

```
1 struct complex {
2     complex(double re, double im)
3     {
4         real = re;
5         imaginary = im;
6     }
7 };
```

- Alternative syntax to initialise variables in constructors

```
1 struct complex
2 {
3     complex(double re, double im) : real{re}, imaginary{im} {}
4 };
```

- A class can have as many constructors as it needs.

CONSTRUCTORS

```
1 struct complex {
2     complex(double re, double im)
3     {
4         real = re;
5         imaginary = im;
6     }
7     complex()
8     {
9         real = imaginary = 0;
10    }
11    double real, imaginary;
12 };
13 ...
14 complex a(3.2, 9.3) ; // C++11 and older
15 complex b{4.3, 1.9} ; // Preferred
16 complex c{} ;
```

- Constructors may be (and normally are) overloaded.
- When a variable is declared, a constructor whose number and type of arguments matches the initialiser expression is implicitly called
- The **default** constructor is the one without any arguments. That is the one invoked when no arguments are given while creating the object.

CONSTRUCTORS

```
1 struct complex
2 {
3     complex(double re, double im)
4     {
5         real = re;
6         imaginary = im;
7     }
8     complex() {}
9     double real{0.};
10    double imaginary{0.};
11 };
12 ...
13 complex a(4.3, 23.09), b;
```

- Member variables can be initialised to "default values" at the point of declaration

CONSTRUCTORS

```
1 struct complex
2 {
3     complex(double re, double im)
4     {
5         real = re;
6         imaginary = im;
7     }
8     complex() {}
9     double real{0.};
10    double imaginary{0.};
11 };
12 ...
13 complex a(4.3, 23.09), b;
```

- Member variables can be initialised to "default values" at the point of declaration
- Member variables not touched by the constructor stay at their default values

CONSTRUCTORS

```
1 struct complex
2 {
3     complex(double re, double im)
4         : real{re}, imaginary{im}
5     {
6     }
7     complex() {}
8     double real{0.};
9     double imaginary{0.};
10 };
11 ...
12 complex a(4.3, 23.09), b;
```

- Member variables can be initialised to "default values" at the point of declaration
- Member variables not touched by the constructor stay at their default values
- Preferred syntax for initialisation of members in a constructor is shown here . This form of initialisation outside the constructor function body is only possible for constructors

FREEING MEMORY FOR USER DEFINED TYPES

```
1 struct darray
2 {
3     double *data = nullptr;
4     size_t sz = 0;
5     darray(size_t N) : sz{N} {
6         data = new double[sz];
7     }
8 }
9
10 auto tempfunc(double phasediff) -> double
11 {
12     // find number of elements
13     darray A{large_number};
14     // do some great calculations
15     return answer;
16 }
```

What happens to the memory? The struct `darray` has a pointer member, which points to dynamically allocated memory

- When the life of the variable `A` ends, the member variables (e.g. the pointer `data`) go out of scope.
- How does one free the dynamically allocated memory attached to the member `data` ?

FREEING MEMORY FOR USER DEFINED TYPES

```
1 struct darray
2 {
3     double *data{nullptr};
4     size_t sz{0};
5     darray(size_t N) : sz{N} {
6         data = new double[sz];
7     }
8     ~darray() {
9         if (data) delete [] data;
10    }
11 };
12
13 auto tempfunc(double phasediff) -> double
14 {
15     // find number of elements
16     darray A{large_number};
17     // do some great calculations
18     return answer;
19 }
```

For any class which explicitly allocates dynamic memory

- We need a function that cleans up all explicitly allocated memory in use, so that we call it for every object whose lifetime is about to end.
- In C++, such functions are called destructors, and have the name `~` followed by the class name.
- Destructors take no arguments, and there is exactly one for each class
- **The destructor is automatically called when a variable expires.** You don't call it explicitly. It is **always** called whenever the scope of an object ends! It is impossible to forget.

DESTRUCTORS

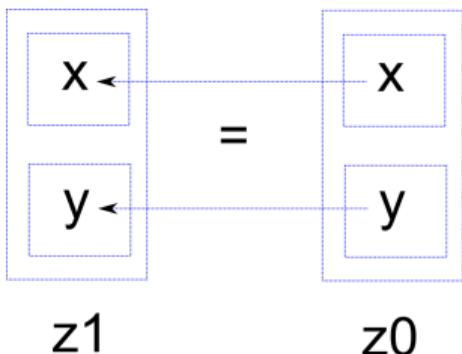
```
1  class A {
2      A() {}
3      ~A() {}
4  };
5  auto demo(A)
6  {
7      A v1;
8      try {
9          A v2;
10         // calc
11
12     } // ~A() for v2
13     catch {
14         // ...
15     }
16 } // ~A() for v1
```

- No matter how you exit a scope, if the scope of a variable ends, its destructor is invoked automatically
- What if we acquire resources in constructors and clean up in the destructor? It would be impossible to forget to free resources when we are done!

COPYING AND ASSIGNMENTS

```
1 struct complex
2 {
3     double x, y;
4 };
5 //...
6 complex z0{2.0, 3.0}, z1;
7 z1 = z0; // assignment operator
8 complex z2{z0}; //copy constructor
```

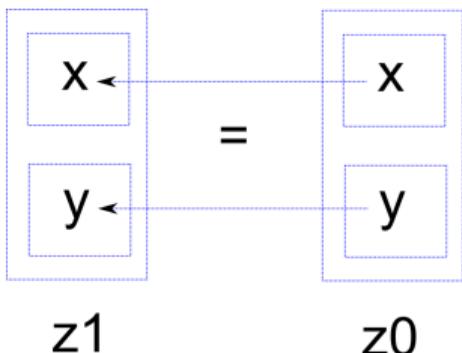
- While copying and assigning, in most cases, we want to assign the data members to the corresponding members



COPYING AND ASSIGNMENTS

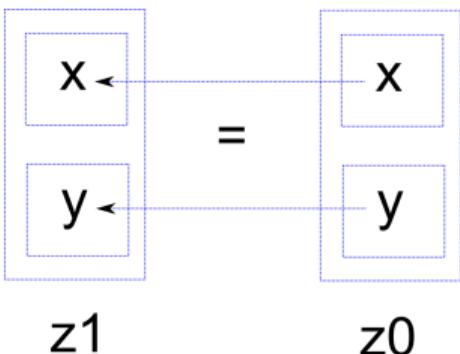
```
1 struct complex
2 {
3     double x, y;
4 };
5 //...
6 complex z0{2.0, 3.0}, z1;
7 z1 = z0; // assignment operator
8 complex z2{z0}; //copy constructor
```

- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations



COPYING AND ASSIGNMENTS

```
1 struct complex
2 {
3     double x, y;
4 };
5 //...
6 complex z0{2.0, 3.0}, z1;
7 z1 = z0; // assignment operator
8 complex z2{z0}; //copy constructor
```

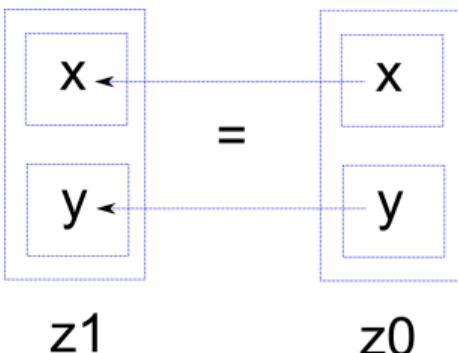


- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations
- You can redefine them for your class

COPYING AND ASSIGNMENTS

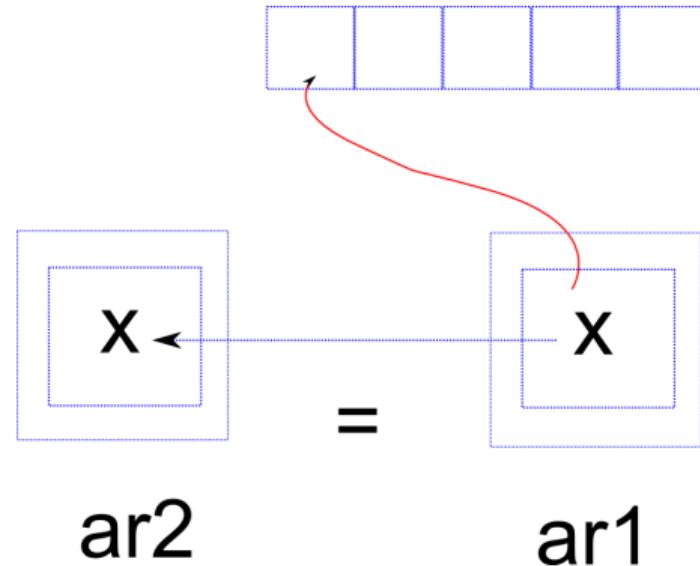
```
1 struct complex
2 {
3     double x, y;
4 };
5 //...
6 complex z0{2.0, 3.0}, z1;
7 z1 = z0; // assignment operator
8 complex z2{z0}; //copy constructor
```

- While copying and assigning, in most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations
- You can redefine them for your class
- Why would you want to ?



COPYING AND ASSIGNMENTS

```
1 class darray {
2     double *x;
3 };
4 darray::darray(unsigned n)
5 {
6     x = new double[n];
7 }
8 void foo()
9 {
10    darray ar1(5);
11    darray ar2{ar1}; //copy constructor
12    ar2[3] = 2.1;
13    //oops! ar1[3] is also 2.1 now!
14 } //trouble
```

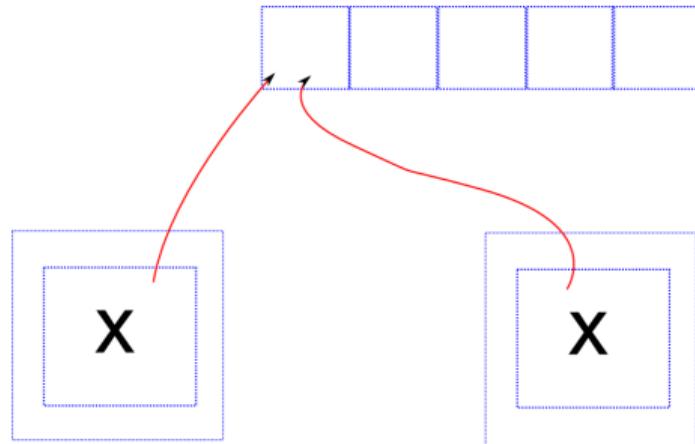


Copying pointers with dynamically allocated memory

- May not be what we want

COPYING AND ASSIGNMENTS

```
1 class darray {
2     double *x;
3 };
4 darray::darray(unsigned n)
5 {
6     x = new double[n];
7 }
8 void foo()
9 {
10    darray ar1(5);
11    darray ar2{ar1}; //copy constructor
12    ar2[3] = 2.1;
13    //oops! ar1[3] is also 2.1 now!
14 } //trouble
```



ar2

ar1

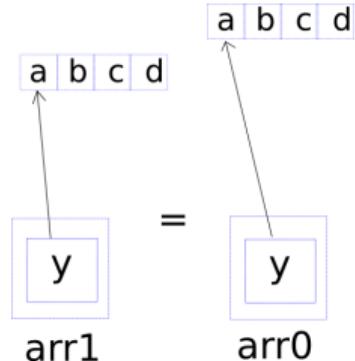
Copying pointers with dynamically allocated memory

- May not be what we want
- Leads to "double free" errors when the objects are destroyed

COPYING AND ASSIGNMENTS

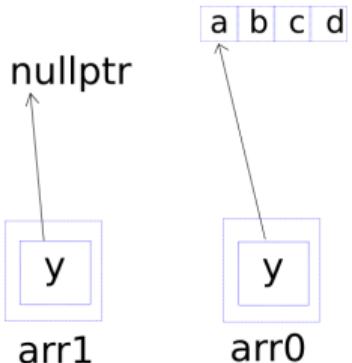
```
1  class darray {
2      double *x{nullptr};
3      unsigned int len{};
4  public:
5      // Copy constructor
6      darray(const darray &);
7      //assignment operator
8      auto operator=(const darray&) -> darray&;
9  };
10 darray::darray(const darray& other)
11 {
12     if (other.len != 0) {
13         len = other.len;
14         x = new double[len];
15         for (unsigned i = 0; i < len; ++i) {
16             x[i] = other.x[i];
17         }
18     }
19 }
20 auto darray::operator=(const darray& other) -> darray&
21 {
22     if (this != &other) {
23         if (len != other.len) {
```

```
1             len = other.len;
2             if (x) delete [] x;
3             x = new double[len];
4         }
5         for (unsigned i = 0; i < len; ++i) {
6             x[i] = other.x[i];
7         }
8     }
9     return *this;
10 }
```



MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

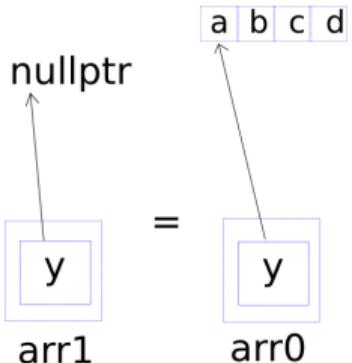
```
1 class darray {
2     darray(darray &&); //Move constructor
3     auto operator=(darray &&) -> darray&;
4     //Move assignment operator
5 };
6 darray::darray(darray&& other)
7 {
8     len = other.len;
9     x = other.x;
10    other.x = nullptr;
11 }
12 auto darray::operator=(darray&& other)
13 -> darray& {
14     len = other.len;
15     x = other.x;
16     other.x = nullptr;
17     return *this;
18 }
19 darray d1(3);
20 init_array(d1); //d1 = {1.0, 2.0, 3.0}
21 darray d2{d1}; //Copy construction
22 // d1 and d2 are {1., 2., 3.}
23 darray d3{std::move(d1)}; //Move
24 // d3 is {1., 2., 3.}, but d1 is empty!
```



- Construct or assign from an R-value reference
(`darray &&`)

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

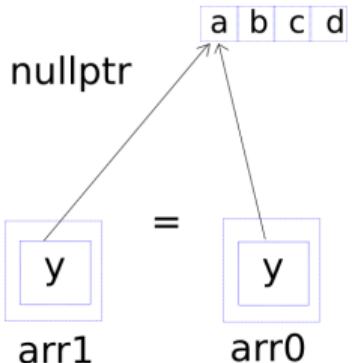
```
1 class darray {
2     darray(darray &&); //Move constructor
3     auto operator=(darray &&) -> darray&;
4     //Move assignment operator
5 };
6 darray::darray(darray&& other)
7 {
8     len = other.len;
9     x = other.x;
10    other.x = nullptr;
11 }
12 auto darray::operator=(darray&& other)
13 -> darray& {
14     len = other.len;
15     x = other.x;
16     other.x = nullptr;
17     return *this;
18 }
19 darray d1(3);
20 init_array(d1); //d1 = {1.0,2.0,3.0}
21 darray d2{d1}; //Copy construction
22 // d1 and d2 are {1.,2.,3.}
23 darray d3{std::move(d1)}; //Move
24 // d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference
(`darray &&`)
- Steal resources from RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

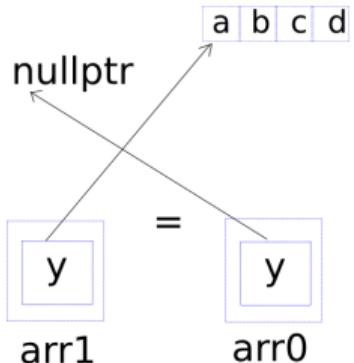
```
1 class darray {
2     darray(darray &&); //Move constructor
3     auto operator=(darray &&) -> darray&;
4     //Move assignment operator
5 };
6 darray::darray(darray&& other)
7 {
8     len = other.len;
9     x = other.x;
10    other.x = nullptr;
11 }
12 auto darray::operator=(darray&& other)
13 -> darray& {
14     len = other.len;
15     x = other.x;
16     other.x = nullptr;
17     return *this;
18 }
19 darray d1(3);
20 init_array(d1); //d1 = {1.0,2.0,3.0}
21 darray d2{d1}; //Copy construction
22 // d1 and d2 are {1.,2.,3.}
23 darray d3{std::move(d1)}; //Move
24 // d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference
(`darray &&`)
- Steal resources from RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

```
1 class darray {
2     darray(darray &&); //Move constructor
3     auto operator=(darray &&) -> darray&;
4     //Move assignment operator
5 };
6 darray::darray(darray&& other)
7 {
8     len = other.len;
9     x = other.x;
10    other.x = nullptr;
11 }
12 auto darray::operator=(darray&& other)
13 -> darray& {
14     len = other.len;
15     x = other.x;
16     other.x = nullptr;
17     return *this;
18 }
19 darray d1(3);
20 init_array(d1); //d1 = {1.0,2.0,3.0}
21 darray d2{d1}; //Copy construction
22 // d1 and d2 are {1.,2.,3.}
23 darray d3{std::move(d1)}; //Move
24 // d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference
(`darray &&`)
- Steal resources from RHS
- Put disposable content in RHS

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference
- Usually you will not be using it explicitly. Results of the evaluation of expressions might create a nameless object containing the resultant value (*prvalue*: pure r-value). A function may be returning a named entity which is about to expire (*xvalue*: expiring value) References to such objects are called R-value references. A move constructor or assignment operator is automatically invoked if constructor argument is an R-value reference

MOVE CONSTRUCTOR/ASSIGNMENT OPERATOR

- You can enable move semantics for your class by writing a constructor or assignment operator using an R-value reference
- Usually you will not be using it explicitly. Results of the evaluation of expressions might create a nameless object containing the resultant value (*prvalue*: pure r-value). A function may be returning a named entity which is about to expire (*xvalue*: expiring value) References to such objects are called R-value references. A move constructor or assignment operator is automatically invoked if constructor argument is an R-value reference
- You can invoke the move constructor by casting the function argument to an R-value reference, e.g.

```
darray d3{std::move(d1)}
```

BIG FIVE (OR ZERO)

- Default constructor
- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator
- How many of these do you have to write for each and every class you make ?

BIG FIVE (OR ZERO)

- Default constructor
- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator
- How many of these do you have to write for each and every class you make ?
- Answer: **None!** If you don't have bare pointers in your class, and don't want anything fancy happening, the compiler will auto-generate reasonable defaults. "Rule of zero"

BIG FIVE

```
1 class darray {
2 public:
3     darray(double x, double y) : re{x}, im{y} {}
4     darray() = default;
5     darray(const darray &) = default;
6     darray(darray &&) = default;
7     auto operator=(const darray&) -> darray& = default;
8     auto operator=(darray&&) -> darray& = default;
9 }
```

- If you have to write any constructor yourself, auto-generation of the default constructor is disabled

BIG FIVE

```
1 class darray {
2 public:
3     darray(double x, double y) : re{x}, im{y} {}
4     darray() = default;
5     darray(const darray &) = default;
6     darray(darray &&) = default;
7     auto operator=(const darray&) -> darray& = default;
8     auto operator=(darray&&) -> darray& = default;
9 }
```

- If you have to write any constructor yourself, auto-generation of the default constructor is disabled
- But you can request default versions of the any of these functions as shown

BIG FIVE

```
1 class darray {
2     darray() = delete;
3     darray(const darray &) = delete;
4     darray(darray &&) = default;
5     auto operator=(const darray &) -> darray& = delete;
6     auto operator=(darray &&) -> darray& = default;
7 }
```

- You can also explicitly request that one or more of these are not auto-generated
- In the example shown here, it will not be possible to copy objects of the class, but they can be moved

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management

```
1  auto operator=(const darray& oth) -> darray& {
2      if (this != &oth) {
3          if (arr && sz != oth.sz) {
4              sz = oth.sz;
5              delete [] arr;
6              arr = new T[sz];
7          }
8          for (size_t i = 0; i < sz; ++i)
9              arr[i] = oth.arr[i];
10     }
11     return *this;
12 }
13 auto operator=(darray&& oth) -> darray& {
14     swap(oth);
15     return *this;
16 }
```

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management
- Pass argument to the assignment operator by value instead of reference

```
1 auto operator=(darray d) -> darray& {
2     swap(d);
3     return *this;
4 }
5 // No further move assignment operator!
```

COPY AND SWAP

- We want to reuse the code in the copy constructor and destructor to do memory management
- Pass argument to the assignment operator by value instead of reference
- Use the class member function `swap` to swap the data with the newly created copy

```
1 auto operator=(darray d) -> darray& {
2     swap(d);
3     return *this;
4 }
5 // No further move assignment operator!
```

- Neat trick that works in most cases
- Reduces the big five to big four

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
1 auto foo(complex a, int p, truck c) -> int
2 {
3     complex z1, z2, z3 = a;
4 ...
5     z1 = z1.argument() * z2.modulus() * z3.conjugate();
6     c.start(z1.imaginary * p);
7 }
```

Imagine that ...

- We have used our complex number structure in a lot of places

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
1 auto foo(complex a, int p, truck c) -> int
2 {
3     complex z1, z2, z3 = a;
4 ...
5     z1 = z1.argument() * z2.modulus() * z3.conjugate();
6     c.start(z1.imaginary * p);
7 }
```

Imagine that ...

- We have used our complex number structure in a lot of places
- Then one day, it becomes evident that it is more efficient to define the complex numbers in terms of the `modulus` and `argument`, instead of the real and imaginary parts.

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
1 auto foo(complex a, int p, truck c) -> int
2 {
3     complex z1, z2, z3 = a;
4 ...
5     z1 = z1.argument() * z2.modulus() * z3.conjugate();
6     c.start(z1.imaginary * p);
7 }
```

Imagine that ...

- We have used our complex number structure in a lot of places
- Then one day, it becomes evident that it is more efficient to define the complex numbers in terms of the `modulus` and `argument`, instead of the real and imaginary parts.
- We have to change a lot of code.

PUBLIC AND PRIVATE MEMBERS

Separating interface and implementation

```
1 auto foo(complex a, int p, truck c) -> int
2 {
3     complex z1, z2, z3 = a;
4 ...
5     z1 = z1.argument() * z2.modulus() * z3.conjugate();
6     c.start(z1.imaginary * p);
7 }
```

Imagine that ...

- External code calling only member functions to access member data can survive
- Direct use of member variables while using a class is often messy, the implementer of the class then loses the freedom to change internal organisation of the class for efficiency or other reasons

C++ CLASSES

```
1 class complex
2 {
3     public:
4         complex(double re, double im)
5             : m_real(re), m_imag(im) {}
6         complex() = default;
7         auto real() const -> double { return m_real; }
8         auto imag() const -> double { return m_imag; }
9     ...
10    private:
11        double m_real = 0., m_imag = 0.;
12 }
```

struct \Rightarrow members public by default
class \Rightarrow members private by default

- Members declared under the keyword **private** can not be accessed from outside
- Public members (data or function) can be accessed
- Provide a consistent and useful interface through public functions
- Keep data members hidden
- Make accessor functions **const** when possible

Exercise 2.1:

Write a toy class for your own experiments. Let's call it `Entry`. An `Entry` is supposed to contain some information about a book, say, name, publication year and anything else you fancy. Keep it minimal. We are only interested in the C++ syntax side of things at this point. Write the necessary member functions to get and set the member variables. Fill in the constructors and the destructor, if you need them. Think about what should or should not be `const` in your member functions!

Exercise 2.2:

The program `examples/complex_number_class.cc` contains a version of the complex number class, with all syntax elements we discussed in the class. It is heavily commented with explanations for every subsection. Please read it to revise all the syntax relating to classes. Write a `main` program to use and test the class.

CONSTRUCTOR/DESTRUCTOR CALLS

Exercise 2.3:

The file `examples/verbose_ctordtor.cc` demonstrates the automatic calls to constructors and destructors. The simple class `Vbose` has one `string` member. All its constructors and destructors print messages to the screen when they are called. The `main()` function creates and uses some objects of this class. Follow the messages printed on the screen and link them to the statements in the program. Does it make sense
(i) When the copy constructor is called ? (ii) When is the move constructor invoked ? (iii) When the objects are destroyed ?

Suggested reading: <http://www.informit.com/articles/printfriendly/2216986>

Exercise 2.4:

The program `examples/onexcept.cc` shows the behaviour of constructor/destructor calls when an exception is called. Observe that exiting a function via an exception is also leaving the scope, and therefore invokes the destructor.

MAKING STD::COUT RECOGNISE CLASS

Teaching cout how to print your type: overload operator `<<`

```
1 auto operator<<(std::ostream& os, const complex& a) -> std::ostream&
2 {
3     os << a.real();
4     if (a.imag() < 0) os << a.imag() << " i ";
5     // If imag() is negative, it already has a - sign
6     else os << " +" << a.imag() << " i ";
7     return os;
8 }
9 complex a;
10 ...
11 std::cout << "The roots are " << a << " and " << a.conjugate() << '\n';
```

AND SIMILARLY FOR STD::CIN

```
1 auto operator>>(std::istream& is, complex& a) -> std::istream&
2 {
3     double x, y;
4     is >> x >> y;
5     a.set_real(x);
6     a.set_imag(y);
7     return is;
8 }
```

- It is up to you to decide IO operations for your classes
- The stream parameters can not be `const`, because by reading from or writing to the stream, we change its state

PRACTISE: WRITE A DATA ROW CLASS

Exercise 2.5:

You now have all the ingredients to write a data row class. A tabular data file has 5 columns. The first two are integers, the rest are doubles. Let's call the columns id, cat, x, y, and z, respectively. Make sure that there are IO stream overloads for the reading and writing objects of that type. Demonstrate by reading a suitable data file "multicolumn.dat", and storing the rows in a vector of your `DataRow` type. You should then be able to sort the vector according to any of the data columns.

USING YOUR CLASS WITH STD::FORMAT AND STD::PRINT

- `std::print` relies on the standard library formatting facilities

- When the compiler parses

```
std::format("the value is {}\\n", v),
```

where `v` is of a type `T`, the compiler looks for the definition of a template specialisation of `std::formatter<T>` with two member functions `parse` and `format`.

- Example, after the discussion of template specialisation!

DATATYPES

Type	Bits	Value
Float	0100 0000 0100 1001 0000 1111 1101 1011	3.1415927
Int	0100 0000 0100 1001 0000 1111 1101 1011	1078530011

- Same bits, different rules \implies different type

From arbitrary collection of members to a new “data type”

```
1 class Date {
2     int m_day, m_month, m_year;
3 public:
4     static auto today() -> Date;
5     auto operator+(int n) const -> Date;
6     auto operator-(int n) const -> Date;
7     auto operator-(const Date &) const -> int;
8 };
```

- Make sure every way to create an object results in a valid state
- Provide only those operations on the data which keep the essential properties intact

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
 - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
 - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.
 - A Date structure could have 3 integers for day, month and year, but they can not be, for example, 0,-1,1

CLASS INVARIANTS

- A class is supposed to represent an idea: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
 - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.
 - A Date structure could have 3 integers for day, month and year, but they can not be, for example, 0,-1,1
- Using `private` data members and well designed `public` interfaces, we can ensure that assumptions behind an idea are always true.

CLASS INVARIANTS

- Construct ensuring class Invariants

```
1  class darray {
2  private:
3      double * dataptr = nullptr;
4      size_t sz = 0;
5  public:
6      // initialize with N elements
7      darray(size_t N);
8      ~darray();
9      // resize to N elements
10     void resize(size_t N);
11     // other members who don't change
12     // dataptr or sz
13 };
```

CLASS INVARIANTS

```
1 class darray {
2     private:
3         double * dataptr = nullptr;
4         size_t sz = 0;
5     public:
6         // initialize with N elements
7         darray(size_t N);
8         ~darray();
9         // resize to N elements
10        void resize(size_t N);
11        // other members who don't change
12        // dataptr or sz
13    };
```

- Construct ensuring class Invariants
- Maintain Invariants in every member

CLASS INVARIANTS

```
1 class darray {
2     private:
3         double * dataptr = nullptr;
4         size_t sz = 0;
5     public:
6         // initialize with N elements
7         darray(size_t N);
8         ~darray();
9         // resize to N elements
10        void resize(size_t N);
11        // other members who don't change
12        // dataptr or sz
13    };
```

- Construct ensuring class Invariants
- Maintain Invariants in every member
- → a structure which always has sensible values

STATIC MEMBERS

```
1 class Triangle {
2 public:
3     static unsigned counter;
4     Triangle() : ...
5     {
6         ++counter;
7     }
8     ~Triangle() { --counter; }
9     static auto instanceCount() -> unsigned
10    {
11        return counter;
12    }
13 };
14 ... Triangle.cc ...
15 unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.

STATIC MEMBERS

```
1 class Triangle {
2 public:
3     static unsigned counter;
4     Triangle() : ...
5     {
6         ++counter;
7     }
8     ~Triangle() { --counter; }
9     static auto instanceCount() -> unsigned
10    {
11        return counter;
12    }
13 };
14 ... Triangle.cc ...
15 unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application

STATIC MEMBERS

```
1 class Triangle {
2 public:
3     static unsigned counter;
4     Triangle() : ...
5     {
6         ++counter;
7     }
8     ~Triangle() { --counter; }
9     static auto instanceCount() -> unsigned
10    {
11        return counter;
12    }
13 };
14 ... Triangle.cc ...
15 unsigned Triangle::counter = 0;
```

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application
- Must be initialised in a source file somewhere, or else you get an "unresolved symbol" error

STATIC MEMBERS

```
1 class Triangle {
2 public:
3     static unsigned counter;
4     Triangle() : ...
5     {
6         ++counter;
7     }
8     ~Triangle() { --counter; }
9     static auto instanceCount() -> unsigned
10    {
11        return counter;
12    }
13 };
14 ... Triangle.cc ...
15 unsigned Triangle::counter = 0;
```

- Static member functions do not have an implicit `this` pointer argument. They can be invoked as
`ClassName::function()`.

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application
- Must be initialised in a source file somewhere, or else you get an "unresolved symbol" error

SOME FUN: OVERLOADING THE () OPERATOR

```
1 class swave
2 {
3     private:
4         double a = 1.0, omega = 1.0;
5     public:
6         swave() = default;
7         swave(double x, double w) :
8             a{x}, omega{w} {}
9         auto operator()(double t) const -> double
10    {
11        return a * sin(omega * t);
12    }
13};
```

```
1 const double pi = acos(-1);
2
3 int N = 100;
4 swave f{2.0, 0.4};
5 swave g{2.3, 1.2};
6
7 for (int i = 0; i < N; ++i) {
8     double ar = 2 * i * pi / N;
9     std::cout << i << " " << f(ar)
10        << " " << g(ar)
11        << '\n';
12}
```

Functionals

- Function like objects, i.e., classes which define a `()` operator

SOME FUN: OVERLOADING THE () OPERATOR

```
1 class swave
2 {
3     private:
4         double a = 1.0, omega = 1.0;
5     public:
6         swave() = default;
7         swave(double x, double w) :
8             a{x}, omega{w} {}
9         auto operator()(double t) const -> double
10    {
11        return a * sin(omega * t);
12    }
13};
```

```
1 const double pi = acos(-1);
2
3 int N = 100;
4 swave f{2.0, 0.4};
5 swave g{2.3, 1.2};
6
7 for (int i = 0; i < N; ++i) {
8     double ar = 2 * i * pi / N;
9     std::cout << i << " " << f(ar)
10        << " " << g(ar)
11        << '\n';
12}
```

Functionals

- Function like objects, i.e., classes which define a `()` operator
- If they return a `bool` value, they are called predicates

FUNCTIONALS

Using function like objects

- They are like other variables. But they can be used as if they were functions!
- You can make vectors or lists of functionals, pass them as arguments ...
- Although you can run any recipe you want by overloading an operator, most operators are limited to one or two arguments. `()` can take as many as you need. This also contributes to functionals looking like functions when in use.

WRITE YOUR OWN FUNCTIONAL!

Exercise 2.6:

Write a functional class where the return value of $f(x)$ is given by a user specified piece-wise continuous linear function. You should write a class PieceWise. It should have a function to read a vector of x_i, y_i values from a file. Sort them according to x values. Then implement an `operator()` function, so that when you write

```
1 PieceWise f;
2 f.read_file("somefile.dat");
3 auto y = f(x);
```

you get the correct piece wise linear function evaluated. Use the standard library function `std::lerp` to perform the linear interpolation.

OVERLOADING OTHER OPERATORS FOR EXPRESSIVE CODE

```
1 // examples/collect.cc
2 class collect {
3     std::vector<int> dat;
4 public:
5     auto operator|(int i) -> collect&
6     {
7         dat.push_back(i);
8         return *this;
9     }
10    auto operator~() const noexcept -> decltype(dat)
11    {
12        return dat;
13    }
14 };
15 auto main() -> int
16 {
17     auto C = collect{};
18     C | 1 | 2 | 3 | 4;
19     for (auto el : (~C)) {
20         std::cout << el << "\n";
21     }
22 }
```

- Operator overloading is not limited to arithmetic and shift operators.
- Sometimes, choosing the right operator to overload can increase the expressiveness of the code

args | sv::drop(1) | sv::transform(str)

USER DEFINED LITERALS

Redefining the """ operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value

```
1 auto main() -> int
2 {
3     double N=6.023e23;
4     Temperature T;
5     T.value(293.0);
6     auto U = Temperature{373.0};
7     auto T2 = 350_C;
8     auto T3 = 900_K;
9     complex c = 1+2_i;
10    ...
11 }
```

USER DEFINED LITERALS

Redefining the """ operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value

```
1 int main()
2 {
3     double N=6.023e23;
4     Temperature T;
5     T.value(293.0);
6     auto U = Temperature(373.0);
7     auto T2 = 350_C;
8     auto T3 = 900_K;
9     complex c = 1+2_i;
10    ...
11 }
```

USER DEFINED LITERALS

Redefining the """ operator!

- You know how to create objects and set their values
- You even know how to construct with a given initial value
- It's far cooler to initialise with your own literals!
- Redefine how literals are interpreted for your class
- Desirable to enable clean and easily read initialisations

```
1 int main()
2 {
3     double N=6.023e23;
4     Temperature T;
5     T.value(293.0);
6     auto T2 = 350_C;
7     auto T3 = 900_K;
8     complex c = 1+2_i;
9     ...
10 }
```

USER DEFINED LITERALS

```
1 auto operator "" _K(long double d) -> Temperature
2 {
3     return { static_cast<double>(d), Temperature::Unit::K };
4 }
5 auto operator "" _C(long double d) -> Temperature
6 {
7     return { static_cast<double>(d), Temperature::Unit::C };
8 }
```

- Defining your own rules for how literals are interpreted for your class
- Desirable to enable clean and easily read initialisations

USER DEFINED LITERALS

Exercise

- The demo program `examples/literals.cc` shows how this is done using a simple “temperature” class
- Make something similar for a `Distance` class!

```
1 auto main() -> int
2 {
3     double N = 6.023e23;
4     auto T2 = 350_C;
5     auto T3 = 900_K;
6 }
```

Inheritance and class hierarchies

CLASS INHERITANCE



Analogy

- Inherited traits: many properties shared among entities of different related types
- Each branch may add new properties
- Seems like a good fit to different ideas we may want to represent in code

CLASS INHERITANCE

```
1 struct Point {double x, y;};
2 class Triangle {
3 public:
4     // Constructors etc., and then,
5     void translate();
6     void rotate(double byangle);
7     auto area() const -> double;
8     auto perimeter() const -> double;
9 private:
10    Point vertex[3];
11 };
12 class Quadrilateral {
13 public:
14     void translate();
15     void rotate(double byangle);
16     auto area() const -> double;
17     auto perimeter() const -> double;
18     auto is_convex() const -> bool;
19 private:
20    Point vertex[4];
21 };
```

Geometrical figures

- Many actions (e.g. translate and rotate) will involve identical code
- Properties like `area` and `perimeter` make sense for all, but are better calculated differently for each type
- There may also be new properties (`is_convex`) introduced by a type

INHERITANCE: BASIC SYNTAX

```
1 class SomeBase {
2     public:
3         double f();
4     protected:
5         int i;
6     private:
7         int j;
8 }
9 class Derived : public SomeBase {
10    void haha() {
11        // can access f() and i
12        // can not access j
13    }
14 }; // Derived is also called a "subtype" of SomeBase
15 void elsewhere()
16 {
17     SomeBase a; // only properties defined in SomeBase
18     Derived b; // b has properties defined in class
19                 // SomeBase and Derived
20     // Can call a.f(),
21     // but e.g., a.i = 0; is not allowed
22 }
```

- Class members can be `private`, `protected` or `public`
- `public` members are accessible from everywhere
- `private` members are for internal use in one class
- `protected` members can be seen by derived classes

INHERITANCE

Base class
data

Derived class
extra data



access of base
class functions



access of derived class functions
(qualified by private, protected etc)

- Inheriting class may add more data, but it retains all the data of the base
- The base class functions, if invoked, will see a base class object
- The derived class object *is a* base class object, but with additional properties

INHERITANCE

Base class
data

Derived class
extra data



access of base
class functions

access of derived class functions
(qualified by private, protected etc)

- A pointer to a derived class always points to an address which also contains a valid base class object.
- `baseptr=derivedptr` is called "upcasting". Always allowed.
- Implicit downcasting is not allowed. Explicit downcasting is possible with `static_cast` and `dynamic_cast`

INHERITANCE

Base class
data

Derived class
extra data



access of base
class functions



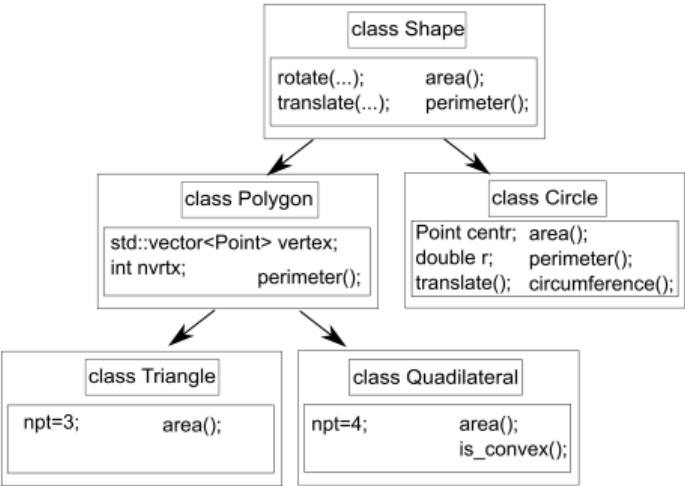
access of derived class functions
(qualified by private, protected etc)

```
1  class Base {
2    public:
3      void f() { std::cout << "Base::f()\n"; }
4    protected:
5      int i{4};
6    };
7  class Derived : public Base {
8    int k{0};
9  public:
10   void g() { std::cout << "Derived::g()\n"; }
11 };
12 auto main() -> int
13 {
14   Derived b;
15   Base *ptr = &b;
16   ptr->g(); // Error!
17   static_cast<Derived *>(ptr)->g(); // OK
18 }
```

CLASS INHERITANCE

- We want to write a program to
 - list the area of all the geometric objects
 - select the largest and smallest objects
 - draw
- in our system.
- A loop over a vector of them will be nice. But `vector< ??? >`
- Object oriented languages like C++, Java, Python ... have a concept of "inheritance" for the classes, to describe such conceptual relations between different types.
- 4 ways to solve this problem in C++ will be introduced at various points in this course

INHERITANCE WITH VIRTUAL FUNCTIONS



- Abstract concept class “Shape”
- Inherited classes add/change some properties
- and inherit other properties from “base” class

A triangle *is* a polygon. A polygon *is* a shape. A circle *is* a shape.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1 class Shape {
2 public:
3     virtual ~Shape() = 0;
4     virtual void rotate(double) = 0;
5     virtual void translate(Point) = 0;
6     virtual auto area() const -> double = 0;
7     virtual auto perimeter() const -> double = 0;
8 };
9 class Circle : public Shape {
10 public:
11     Circle(); // and other constructors
12     ~Circle();
13     void rotate(double phi) {}
14     auto area() const -> double override
15     {
16         return pi * r * r;
17     }
18 private:
19     double r;
20 };
```

- Circle is a derived class from base class Shape
- A derived class inherits from its base(s), which are indicated in the class declaration.
- Functions marked as `virtual` in the base class *can be re-implemented* in a derived class.

Note: In C++, member functions are not virtual by default.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1 class Shape {
2 public:
3     virtual ~Shape() = 0;
4     virtual void rotate(double) = 0;
5     virtual void translate(Point) = 0;
6     virtual double area() const = 0;
7     virtual auto perimeter() const -> double = 0;
8 }
9 class Circle : public Shape {
10 public:
11     Circle(); // and other constructors
12     ~Circle();
13     void rotate(double phi) {}
14     auto area() const -> double override
15     {
16         return pi * r * r;
17     }
18 private:
19     double r;
20 };
21 Shape a; // Error!
22 Circle b; // ok.
```

- A derived class inherits all member variables and functions from its base.
- `virtual` re-implemented in a derived class are said to be "overridden", and ought to be marked with `override`
- A class with a `pure virtual` function (with "`= 0`" in the declaration) is an `abstract` class. Objects of that type can not be declared.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1 class Polygon : public Shape {
2 public:
3     auto perimeter() const -> double final
4     {
5         // return sum over sides
6     }
7 protected:
8     vector<Point> vertex;
9     int npt;
10 };
11 class Triangle : public Polygon {
12 public:
13     Triangle() : npt(3)
14     {
15         vertex.resize(3); // ok
16     }
17     auto area() const -> double override
18     {
19         // return sqrt(s*(s-a)*(s-b)*(s-c))
20     }
21 };
```

Syntax for inheritance

- Triangle implements its own `area()` function, but can not implement a `perimeter()`, as that is declared as `final` in `Polygon`. This is done if the implementation from the base class is good enough for intended inheriting classes.

CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1 class Polygon : public Shape {
2 public:
3     auto perimeter() const -> double final
4     {
5         // return sum over sides
6     }
7 protected:
8     vector<Point> vertex;
9     int npt;
10 };
11 class Triangle : public Polygon {
12 public:
13     Triangle() : npt(3)
14     {
15         vertex.resize(3); // ok
16     }
17     auto area() -> double override // Error!!
18     {
19         // return sqrt(s*(s-a)*(s-b)*(s-c))
20     }
21 };
```

- The keyword `override` ensures that the compiler checks there is a corresponding base class function to override.
- Virtual functions can be re-implemented without this keyword, but an accidental omission of a `const` or an `&` can lead to really obscure runtime errors.

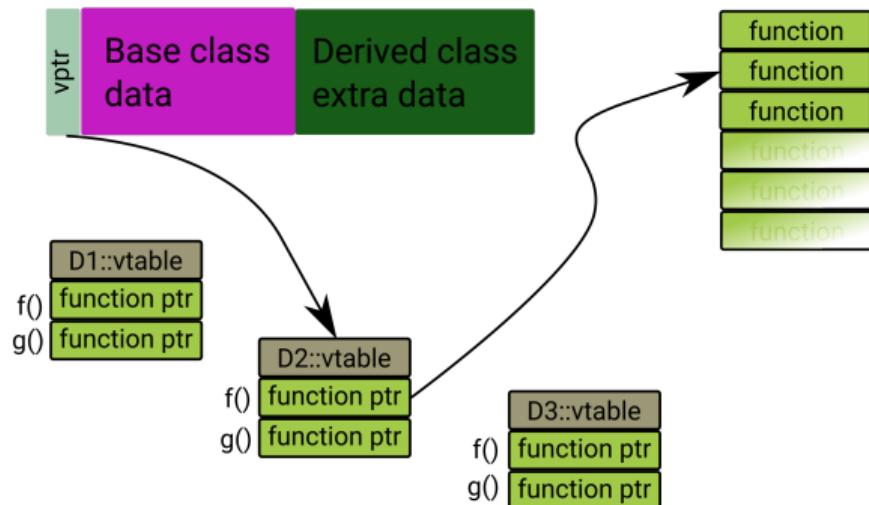
CLASS INHERITANCE WITH VIRTUAL FUNCTIONS

```
1 auto main() -> int
2 {
3     vector<std::unique_ptr<Shape>> shape;
4     shape.push_back(std::make_unique<Circle>(0.5, Point(3,7)));
5     shape.push_back(std::make_unique<Triangle>(Point(1,2), Point(3,3), Point(2.5,0)));
6 ...
7     for (size_t i = 0;i < shape.size(); ++i) {
8         std::cout << shape[i]->area() << '\n';
9     }
10 }
```

- A pointer to a base class is allowed to point to an object of a derived class
- Here, `shape[0]->area()` will call `Circle::area()`, `shape[1]->area()` will call `Triangle::area()`

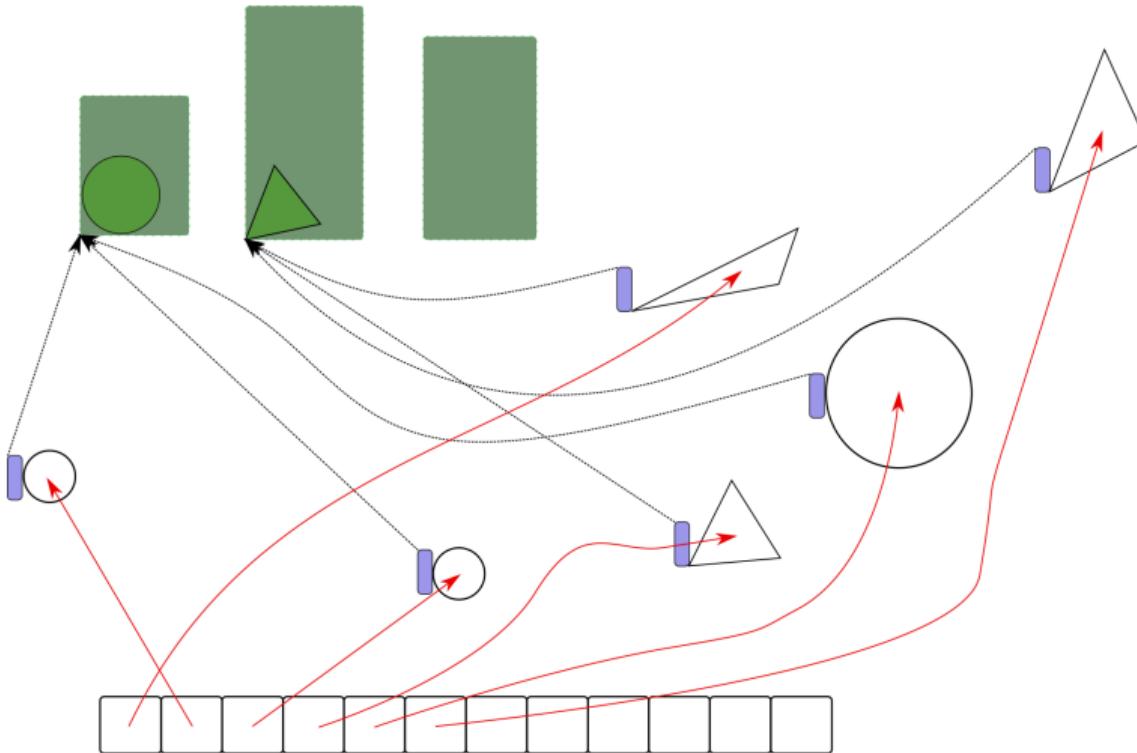
CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

```
D *d=new D2;  
d->f();
```



- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional book keeping code
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere
- The `vptr` pointer points to the `vtable` of that particular class

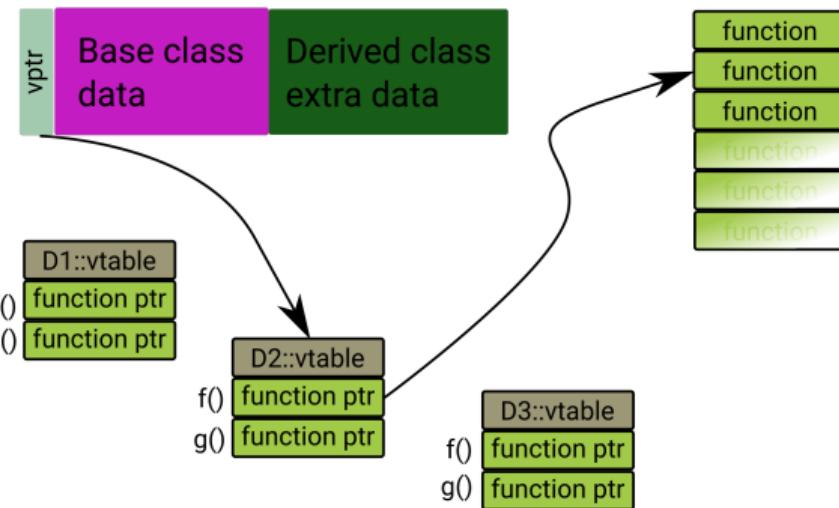
CALLING VIRTUAL FUNCTIONS: HOW IT WORKS



CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual!** The overhead, with modern machines and compilers, is not huge. But abusing this feature **will hurt performance**

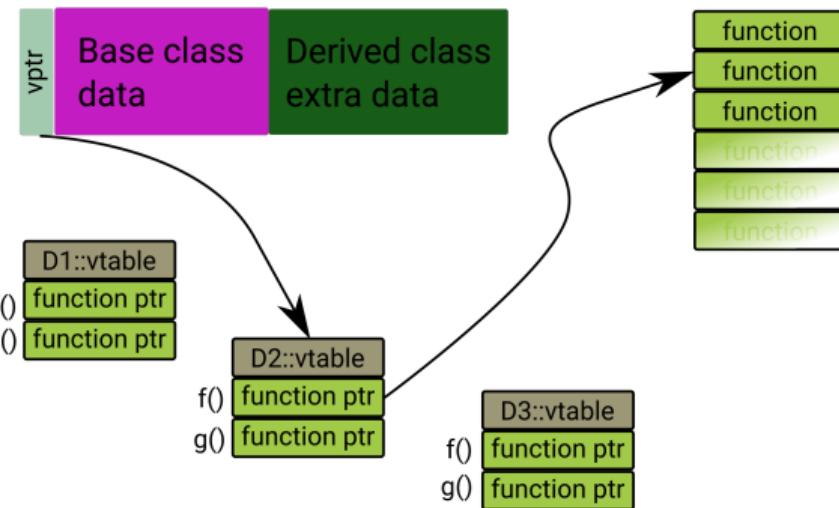
```
D *d=new D2;  
d->f();
```



CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

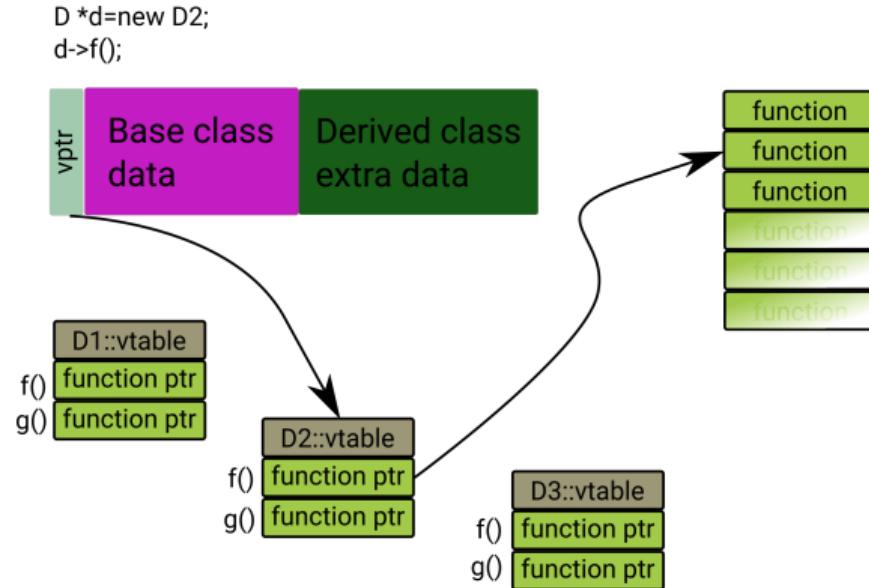
- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual!** The overhead, with modern machines and compilers, is not huge. But abusing this feature **will hurt performance**

```
D *d=new D2;  
d->f();
```



CALLING VIRTUAL FUNCTIONS: HOW IT WORKS

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual!** The overhead, with modern machines and compilers, is not huge. But abusing this feature **will hurt performance**



- But if virtual functions offer the cleanest solution with acceptable performance, **don't invent weird things to avoid them!**

CLASS INHERITANCE

Inherit or include as data member ?

```
1 class DNA {  
2 ...  
3     std::valarray<char> seq;  
4 };  
5 class Cell : public DNA ???  
6 or  
7 class Cell {  
8 ...  
9     DNA mydna;  
10};
```

- A derived class **extends** the concept represented by its base class in some way.
- Although this extension might mean addition of new data members,

$$B = A \oplus \text{newdata}$$

does not necessarily mean the class for B should inherit from the class for A

CLASS INHERITANCE

Inherit or include as data member ?

```
1 class DNA {  
2 ...  
3     std::valarray<char> seq;  
4 };  
5  
6 class Cell : public DNA ???  
7  
8 or  
9  
10 class Cell {  
11 ...  
12     DNA mydna;  
13 };  
14
```

is vs has

- A good guide to decide whether to inherit or include is to ask whether the concept B **contains** an object A, or whether any object of type B **is** also an object of type A, like a monkey **is** a mammal, and a triangle **is** a polygon.
- **is** \Rightarrow inherit . **has** \Rightarrow include

CLASS INHERITANCE

Inheritance summary

- Base classes to represent common properties of related types : e.g. all proteins are molecules, but all molecules are not proteins. All triangles are polygons, but not all polygons are triangles.
- Less code: often, only one or two properties need to be changed in an inherited class
- Helps create reusable code
- A base class may or may not be constructible (Polygon as opposed to Shape)

CLASS DECORATIONS

More control over classes

- Possible to initialise data in class declaration
- Initialiser list constructors
- Delegating constructors allowed
- Inheriting constructors possible

```
1  class A {
2      int v[]{1, -1, -1, 1};
3  public:
4      A() = default;
5      A(std::initializer_list<int> &);
6      A(int i, int j, int k, int l)
7      {
8          v[0] = i;
9          v[1] = j;
10         v[2] = k;
11         v[3] = l;
12     }
13     //Delegate work to another constructor
14     A(int i, int j) : A(i, j, 0, 0) {}
15 };
16 class B : public A {
17 public:
18     // Inherit all constructors from A
19     using A::A;
20     B(string s);
21 };
```

MORE CONTROL OVER CLASSES

- Explicit `default`, `delete`, `override` and `final`
- "Explicit is better than implicit"
- More control over what the compiler does with the class
- Compiler errors better than hard to trace run-time errors due to implicitly generated functions

```
1  class A {  
2      // Automatically generated is ok  
3      A() = default;  
4      // Don't want to allow copy  
5      A(const A &) = delete;  
6      A & operator=(const A &) = delete;  
7      // Instead, allow a move constructor  
8      A(const A &&);  
9      // Don't try to override this!  
10     void getDrawPrimitives() final;  
11     virtual void show(int i);  
12 };  
13 class B : public A  
14 {  
15     B() = default;  
16     void show() override; //will be an error!  
17 };  
18 
```

Exercise 2.7:

The directory `exercises/geometry` contains a set of files for the classes Point, Shape, Polygon, Circle, Triangle, and Quadrilateral. In addition, there is a `main.cc` and a `CMakeLists.txt`. Observe the use of the keywords like `default`, `override`, `final` etc. Familiarise yourself with

- Implementation of inherited classes
- Compiling multi-file projects
- The use of base class pointer arrays to work with heterogeneous types of objects

```
mkdir build
cd build
CXX=g++ cmake ..
make
```
