COMP-3400-30-R-2024W C++, Week 3

Object-Oriented Programming

Mr. Paul Preney

School of Computer Science, University of Windsor, Windsor, Ontario

Not all slides will necessarily be presented in the lecture. It is your responsibility to read and learn all slide content and related materials in the textbooks and in this course.

2026-01-20

Outline

Inheritance Overview

C++ support for object-based/oriented programming includes:

- support for both single and multiple inheritance of any class/struct
- each type can be **`virtual`**ly or non-**`virtual`**ly inherited from
- when inheriting from a type, a scope access applied to the inherited class' members, i.e., **`public`**, **`protected`**, **`private`**
- unless inheritance is explicitly specified, C++ **`class`**es and **`struct`**ures do *not* inherit from any other type
- **`union`**s are not permitted to inherit from other types
- when porting code from other object-oriented languages, inheritance code should be **`virtual public`** in C++

Member Overview

- **`static`** member variables/functions are like global variable/functions except their scope is constrained to be within the type they are defined in
- when porting code from other object-oriented languages, all non-**`static`** member functions should be **`virtual`** in C++

Member Overview (con't)

Every C++ **`struct`**, **`union`**, and **`class`**, can have special members:

- a default constructor
- a copy constructor
- a move constructor (starting in C++11)
- a copy assignment operator
- a move assignment operator (starting in C++11)
- a destructor

Constructors

All C++ special members that are constructors:

- have **no return type**
- have the same name as the name of the `class`, `struct`, or `union` they are defined in
- are not considered functions in the language
  - e.g., they can only be invoked by creating an instance of that type

Constructors (con't)

w03/constructor-examples.cxx

```
#include <utility>
struct Type
{
  int* p;

  Type() : p(new int(42)) { }                    // default constructor
  Type(int i) : p(new int(i)) { }                // a constructor
  Type(Type const& t) : p(new int(*t.p)) { }     // copy constructor
  Type(Type&& t) : p(t.p) { t.p = nullptr; }     // move constructor

  ~Type() { delete p; }                          // destructor
};

Type a;                                          // a is default constructed
Type b=a, c(a);                                  // b and c are copy constructed
Type d=std::move(a), e(std::move(b)), f(Type());  // d, e, and f are move constructed
Type y(59), z(60);                               // y and z invoke the int constructor
```

Constructors (con't)

A **default constructor** is a constructor:

- with **no arguments,** and,
- is invoked when an object is created having no arguments passed to its constructor.

Constructors (con't)

w03/default-constructor.cxx

```
1  struct Type
2  {
3    int* p;
4    // ...
5    Type() :              // invoke constructors and initialize members after : before {
6      p(new int(42))      // construct int* with the result of new int(42)
7    {                     // after { all members have been constructed
8    }
9    // ...
10 };
```

Constructors (con't)

A **copy constructor** is a constructor:

- with a T `const`& argument, and,
  - where T is the name of the `class`, `struct`, or `union` it is in
- is invoked when the compiler/programmer needs to create and copy the data of another instance of the same type into the created type.

Constructors (con't)

w03/copy-constructor.cxx

```
1  struct Type
2  {
3    int* p;
4    //  ...
5    Type(Type const& t) :   // invoke constructors and initialize members after : before {
6      p(new int(*t.p))      // construct int* with the result of new int(42)
7    {                       // after { all members have been constructed
8    }
9    // ...
10 };
```

Constructors (con't)

A **move constructor** is a constructor:

- with a T&& argument, and,
  - where T is the name of the **`class`**, **`struct`**, or **`union`** it is in
- is invoked when the compiler/programmer needs to create and move the data of another instance of the same type into the created type.

Constructors (con't)

w03/move-constructor.cxx

```
1  struct Type
2  {
3    int* p;
4    //  ...
5    Type(Type&& t) :          // invoke constructors and initialize members after : before {
6      p(t.p)                  // construct int* with the result of new int(42)
7    {                         // after { all members have been constructed
8      t.p = nullptr;          // t refers to an object that still needs to be destroyed!
9    }
10   // ...
11 };
```

Destructors

All destructors:

- have **no return type**
- have their name prefixed with a ~ in front of the **`class`**', **`struct`**'s, or **`union`**'s name they are defined in
- *are* considered **`void`** functions in the language
  - **Never** invoke a destructor directly unless using "placement **`new`**"!
- should not emit exceptions
- are implicitly **`noexcept`** unless explicitly stated otherwise in code (starting with C++11)
- are assumed by C++ programmers to **release all owned resources** acquired during the lifetime of an object
  - e.g., calling **`operator delete`**, `close(fd)`, etc. as is needed
  - i.e., "Resource Acquisition is Initialization" (RAII)

Destructors (con't)

- are invoked when an instance object goes out-of-scope
  - e.g., global variables, stack-allocated (i.e., automatic) variables, members in an aggregate type that is being destructed, etc.
- are invoked when the appropriate **`operator delete`** or **`operator delete[]`** is called on dynamically-allocated object
- are explicitly invoked when a programmer needs to destroy an object created with "placement **`new`**"

Destructors (con't)

w03/destructor.cxx

```
1  struct Type
2  {
3    int* p;
4    // ...
5    ~Type()
6    {
7      delete p;                          // free memory associated with p
8    }
9    // ...
10 };
```

Copy and Move Assignment Operators

A **copy assignment operator**:

- is a member assignment operator overload
- with a single T `const`& argument.

Copy and Move Assignment Operators (con't)

w03/copy-assignment.cxx

```
1  #include <utility>                    // for std::swap since C++11, <algorithm> before C++11
2  struct Type {
3    int* p;
4    Type(int const& i) : p(new int{i}) { }
5    Type(Type const& t) : p(new int{*t.p}) { }
6    Type& operator =(Type const& t)     // copy assignment operator
7    {
8      Type local(t);                    // create a local copy of t
9      std::swap(p,local.p);             // exchange pointers
10     return *this;                     // local will now be destroyed
11   }
12 };
13 int main()
14 {
15   Type a(42), b{52}, c = 62;          // a and b invoke Type(int const&) constructor
16   a = b = c;                          // copy assignment
17 }
```

Copy and Move Assignment Operators (con't)

A **move assignment operator**:

- is a member assignment operator overload
- with a single T&& argument.

Copy and Move Assignment Operators (con't)

w03/move-assignment.cxx

```
#include <utility>
struct Type {
  int* p;
  Type() : p{new int{}} { }
  Type(int const& i) : p{new int{i}} { }
  Type(Type&& t) : p{std::move(t.p)} { }
  Type& operator =(Type&& t)        // move assignment operator
  {
    Type local(std::move(t));       // move t into *this
    std::swap(p,local.p);           // exchange pointers
    return *this;                   // local will now be destroyed
  }
};
int main()
{
  Type a(11), b{12}, c = 13;        // a, b, and c invoke Type(int const&) constructor
  a = std::move(c); b = Type{};     // move assignments
}
```

Outline

Objects & Memory Layout: Single `struct`/`class` Layout

Use of `virtual`: None

References: [5, §2.5.2], [3, [intro.memory], [basic.types]]

Assuming only one access level:

- Each non-`static` data member **appears** in memory in declaration order where subsequent members are at a higher memory address than the previous
  - for data members at the same access level
  - such data members may not be positioned immediately after each other
- The compiler can add **padding** between/after members for **alignment** purposes.
- **Excluding** the **extra space** added for **alignment** purposes, the **size** of the object is the **sum of the sizes** of all data members or 1, whichever is larger.
- In C++11, alignment [3, [basic.align]] can be queried and controlled via `alignas` [3, [dcl.type.simple]], `alignof` [3, [expr.alignof]], `align` [3, [ptr.align]], and, `alignment_of` [3, [meta.unary.prop.query]].

Objects & Memory Layout: Single `struct`/`class` Layout (con't)

Example:

w03/layout-ega.cxx

```
1  struct node            // no inheritance
2  {
3    void add(char);      // not virtual
4    char data_[10];
5    void remove(char);   // not virtual
6    node *next_;
7  };
```

e.g., $\texttt{sizeof(node)} \geq \texttt{sizeof(char[10])} + \texttt{sizeof(node*)}$

e.g., $\texttt{sizeof(node)} = \texttt{sizeof(char[10])} + \mathit{padding}_{\texttt{char[10]}} + \texttt{sizeof(node*)} + \mathit{padding}_{\texttt{node*}}$

`struct`/`class` Layout: Virtual Member Implications

Use of `virtual`: With at least one member function.

References: [5, §3.5.1]

`struct`/`class` layout is determined by the **data members** and the need to have a **compiler-generated table of member function pointers**.

The functions the compiler stores in the table are the ones declared as `virtual`.

The compiler decides:

- **how** it chooses to represent, store, and access the table of function pointers, and,
- **where** it chooses to place the pointer to such (i.e., at the beginning or the end) within the `struct`/`class`, etc.

`struct`/`class` Layout: Virtual Member Implications (con't)

Note:

- Constructors and **`static`** functions cannot be declared as **`virtual`**.
- A **`virtual`** destructor is allocated a slot. All descendants types' destructors will share the same slot as there can only be one destructor per type.
- Compiler **will implicitly add code** to types' (implicit and/or explicit) constructors to properly deal with any required internal state.

`struct`/`class` Layout: Virtual Member Implications (con't)

One way to implement **`virtual`** members:

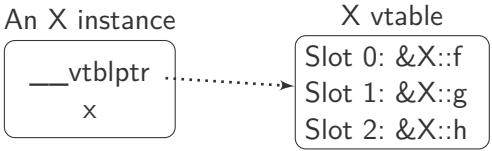
- For each **`struct`**/**`class`** with at least one **`virtual`** function store a pointer to an **array of function pointers** to those **`virtual`** functions in each instance.
  - ASIDE: Since C++ is statically-typed so all types are known at compile time, thus, the compiler *does not* need to store a vtable pointer with each *instance* of a type —it can instead "hard-code" such where needed since it "knows" the type being used in the program code.

struct/class Layout: Virtual Member Implications (con't)

Example X:

w03/layout-egbX.cxx

```
1  struct X
2  {
3    int x;
4    virtual void f();
5    virtual void g(char);
6    virtual void h(float);
7  };
```



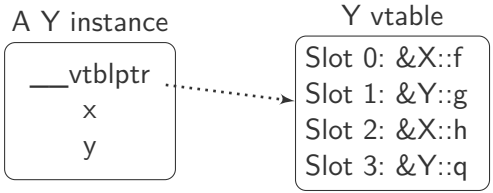
NOTE: One vtable is needed per type.

struct/class Layout: Virtual Member Implications (con't)

Example Y:

w03/layout-egbY.cxx

```
1  // struct X from previous example
2  struct Y : X
3  {
4    int y;
5    void g(char) override;
6    virtual void q(Y const&);
7  };
```

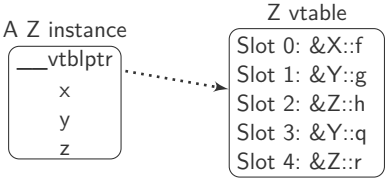


struct/class Layout: Virtual Member Implications (con't)

Example Z:

w03/layout-egbZ.cxx

```
1  // struct X from previous example
2  // struct Y from previous example
3  struct Z : Y
4  {
5    int z;
6    void h(float) override;
7    virtual void r(Z&);
8  };
```

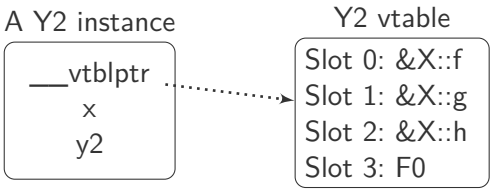


struct/class Layout: Virtual Member Implications (con't)

Example Y2:

w03/layout-egbY2.cxx

```
1  // struct X from previous example
2  struct Y2 : X
3  {
4    int y2;
5    using X::g; // Unhide X members named g
6    virtual void g(double); // F0. Different signature!
7  };
```

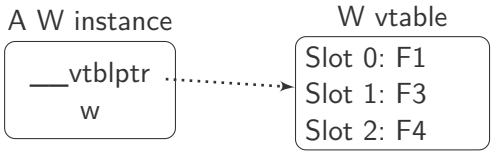


struct/class Layout: Virtual Member Implications (con't)

Example W:

w03/layout-egbW.cxx

```
1  struct W
2  {
3    int w;
4    virtual void f(); // F1
5    void f(long); // F2
6    virtual void f(int); // F3
7    virtual void f(double); // F4
8  };
```



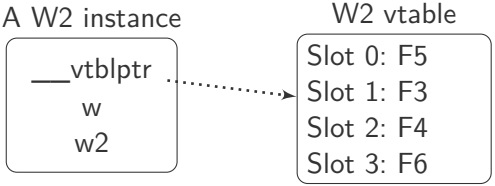
NOTE: Signature matters! F2 is **not** virtual!

struct/class Layout: Virtual Member Implications (con't)

Example W2:

w03/layout-egbW2.cxx

```
1  // struct W from previous example
2  struct W2 : W
3  {
4    int w2;
5    void f() override; // F5
6    void f(long double d); // F6
7  };
```



struct/class Layout: Non-Virtual Inheritance

The memory layout of single-access level, non-**virtual**, **singly-inherited types** is the **concatenation** of the **memory layouts** of all types in the hierarchy **starting with the base type** of the hierarchy:

w03/layout-egc1a.cxx

```
1 struct W { };
2 struct X : W { int x; };
3 struct Y : X { int y; };
4 struct Z : Y { int z; };
```

struct/class Layout: Non-Virtual Inheritance (con't)

In terms of memory layout Z is equivalent to:

w03/layout-egc1b.cxx

```
1 struct Z_mem_layout
2 {
3   int x;
4   int y;
5   int z;
6 };
```

`struct`/`class` Layout: Non-Virtual Inheritance (con't)

Notice that `W` from two slides earlier requires zero (0) bytes of memory and so it is an **empty base class** with respect to X, Y, and Z.

- A C++ compiler will render **any empty base class** in a **single-inheritance hierarchy** so that it **occupies zero bytes of memory**.
- This is often referred to as **empty base optimization** (EBO).
- Otherwise such types will **occupy at least one byte of memory** since the **`sizeof`** of **a type must at least be one (1)**. [3, [expr.sizeof], [class]]

`struct`/`class` Layout: Non-Virtual Inheritance (con't)

The memory layout of single-access level, non-**`virtual`**, **multiply-inherited types** is the **concatenation** of the **memory layouts** of all types in the hierarchy **in some *unspecified* order the inheritance** starting with the root types [3, [class.mi]] of the hierarchies, e.g.,

w03/layout-egd1a.cxx

```
1 struct Z { int z; };
2 struct Y { int y; };
3 struct A : Z { int a; };
4 struct B : A { int b; };
5 struct C : A, Y { int c; };
6 struct D : B, C { int d; };
```

`struct`/`class` Layout: Non-Virtual Inheritance (con't)

In terms of memory layout `D` could be equivalent to:

w03/layout-egd1b.cxx

```
1 struct D_mem_layout_wont_compile
2 {
3   int z; int a; int b; // D::B path
4   int z; int a; int y; int c; // D::C path
5   int d;
6 };
```

struct/class Layout: Non-Virtual Inheritance (con't)

Unqualified access to names that are **ambiguous through inheritance** are **compile-time errors**.

Use the **scope resolution operator** (i.e., `::`) to **qualify** the name by **specifying the type** to **resolve the ambiguity**:

w03/layout-egd1c.cxx

```
1 D d;
2 d.z = 1; // ERROR! Ambiguous!
3 d.B::z = 1; // Okay. Unique via B.
4 d.C::z = 2; // Okay. Unique via C.
5 d.y = 3; // Okay. Unambiguous.
```

`struct`/`class` Layout: Virtual Inheritance

The memory layout of **`virtual`ly inherited types** must ensure that the **inheriting type** will **only** have a **single instance** of each **`virtual`ly inherited type** at that level otherwise all other aspects are the same as non-virtual inheritance.

Consider:

w03/layout-ege1a.cxx

```
1 // Only A is inherited virtually below!
2 struct Z { int z; };
3 struct Y { int y; };
4 struct A : Z { int a; };
5 struct B : virtual A { int b; };
6 struct C : virtual A, Y { int c; };
7 struct D : B, C { int d; };
```

`struct`/`class` Layout: Virtual Inheritance (con't)

In terms of **data members** —not memory layout— D is equivalent to:

w03/layout-ege1b.cxx

```
1  struct D_data_members
2  {
3    int z; int a; // D::A
4    int b; // D::B
5    int y; int c; // D::C
6    int d; // D
7  };
```

since in the hierarchy both B and C **`virtual`**ly inherit from A and **both B and C are siblings at that level in the hierarchy**.

IMPORTANT: There is only **one instance of A**.

`struct`/`class` Layout: Virtual Inheritance (con't)

Compilers have some **discretion** on how exactly to lay out memory.

- The memory layout chosen **must be invariant across all possible uses**!

This problem can be solved by **storing "pointer" offset(s) to the virtually inherited base type(s)**.

struct/class Layout: Virtual Inheritance (con't)

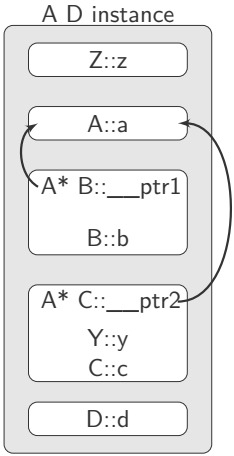
Consider again:

w03/layout-ege1a.cxx

```
1  // Only A is inherited virtually below!
2  struct Z { int z; };
3  struct Y { int y; };
4  struct A : Z { int a; };
5  struct B : virtual A { int b; };
6  struct C : virtual A, Y { int c; };
7  struct D : B, C { int d; };
```

A possible (RAM inefficient) memory layout might be...

struct/class Layout: Virtual Inheritance (con't)



`struct`/`class` Layout: Virtual Inheritance (con't)

Does multiple **`virtual`** inheritance look complicated?

This is **one of the reasons** why most OOP languages **don't support *general* multiple inheritance**.

Some OOP languages **do support *limited* multiple inheritance**, e.g.,

- **no data members**,
- **declared but not defined function members**, and/or
- **all inheritance** must be **`virtual`** which eliminates some layout and access issues.

`struct`/`class` Layout: Virtual Inheritance (con't)

Unqualified access to names that are **ambiguous through inheritance** are **compile-time errors**.

Since **`virtual` inheritance** ensures there is **only one copy of a base class**, accesses that would have been ambiguous with non-virtual inheritance **may** become unambiguous...

struct/class Layout: Virtual Inheritance (con't)

Given:

w03/layout-ege1a.cxx

```
1  // Only A is inherited virtually below!
2  struct Z { int z; };
3  struct Y { int y; };
4  struct A : Z { int a; };
5  struct B : virtual A { int b; };
6  struct C : virtual A, Y { int c; };
7  struct D : B, C { int d; };
```

`struct`/`class` Layout: Virtual Inheritance (con't)

Then there are no issues with this:

w03/layout-ege1c.cxx

```
1 D d;
2 d.z = 1;
3 d.a = 2;
4 d.y = 3;
```

But realize that **in the `worst case` accesses to A's and Z's members** through B, C, and/or D are now **via an indirection/computed memory offset**.

`struct`/`class` Layout: Virtual Inheritance (con't)

Know that **multiple `virtual` inheritance *can* introduce ambiguities**, e.g.,

- when constructing **`virtual`** base objects
- when calling member functions defined in or accessible via the **`virtual`** base from a derived type

struct/class Layout: Virtual Inheritance (con't)

Ambiguous Construction Case...

When there is only one **virtual** base instance for **two or more types**, which type pathway should initialize the **virtual** base instance?

Consider:

w03/layout-ege1d.cxx

```
1 struct A { /* ... */ };
2 struct B : virtual A { /* ... */ };
3 struct C : virtual A, Y {  /* ... */ };
4 struct D : B, C { /* ... */ };
```

Q. When creating an instance of D, **should** B's or C's constructor **initialize** A?

A. In general the compiler **cannot determine** which constructor should be used as it is **ambiguous**.

struct/class Layout: Virtual Inheritance (con't)

w03/layout-ege1e.cxx

```
1  struct A {
2    A(int a) : a_(a) { }
3    int a_;
4  };
5  struct B : virtual A {
6    B(int b) : A(b+1), b_(b) { }
7    int b_;
8  };
9  struct C : virtual A {
10   C(int c) : A(c-1), c_(c) { }
11   int c_;
12 };
13 struct D : B, C {
14   D(int d) : B(d*2), C(d/2), d_(d) { } // UH OH! ERROR!
15   int d;
16 };
```

`struct`/`class` Layout: Virtual Inheritance (con't)

Solution: The programmer must **explicitly invoke** the **`virtual`** **base constructor** in the class **where it is ambiguous**:

w03/layout-ege1f.cxx

```
1  // struct A, B, and C as before
2  struct D : B, C
3  {
4    D(int d) :
5      A(d*2+1), B(d*2), C(d/2), d_(d)
6    {
7    }
8    int d;
9  };
```

i.e., invoking A's constructor in D's initialization list resolves the ambiguity.

NOTE: Note that A's **`int`** is not a value that B's or C's constructor would have set it to!

Ambiguous Member Function Case...

This case is resolved in one of two ways:

- by **qualifying** the call using the **scope resolution operator (::)** —done the same way as qualified data member access, or,
- by **overriding** the member function **in the type where it is ambiguous** and setting the code for that function to properly invoke the correct base/bases member function(s).

`struct`/`class` Layout: In General

C++ permits **all combinations** of `virtual` and non-`virtual` member functions *and* **all combinations** of `virtual` and non-`virtual` inheritance, if any, **regardless of access permissions** (i.e., `public`, `protected`, or `private`) when writing `struct`s or `class`es.

- **NOTE:** `virtual private` member functions are possible *and* **they can be overridden in the derived type** since they occupy a slot.

`struct`/`class` Layout: In General (con't)

When **`virtual`** inheritance and **`virtual`** member functions are used together, this further complicates the memory layout of the type resulting in **layouts** that **combine the techniques previously discussed**.

To **avoid wasting memory in each object instance**, C++ compilers typically will **store all/most type information** (e.g., virtual base offsets, virtual member function tables, and any other type-related information) via a "**`__vtableptr`**" discussed earlier.

`struct`/`class` Layout: In General (con't)

If C++ compilers can **determine** that a **specific** virtual function or data member in some virtual base will **always be directly called**, then it will emit code to directly call/access it.

Advice: If you are **not** writing **run-time polymorphic** code, then ask yourself why you are using `virtual` in your code.

`struct`/`class` Layout: Overhead

Using **`virtual` inheritance** requires **at least one additional indirection** relative to non-**`virtual`** inheritance to access **any member accessible via the `virtual` base type**.

One (1) additional indirection, i.e., the indirect function call itself, is needed if the offset is **not** stored in the vtable.

- **Pro:** Fast since member offset pointer is extremely likely to be on the same L1 cache page as the instance.
- **Con:** RAM to store the offsets **in each instance** plus the constructor overhead to set such **in each instance**.

struct/class Layout: Overhead (con't)

Two (2) additional indirections are needed if the offset is stored in the vtable.

- **Pro:** Only one vtable pointer in each instance is required: constructors need only set one pointer.
- **Con:** Extra cache page and indirection.
  - The first indirection is to access the vtable. The vtable might not be close to the instance data and might be on another cache page. (The required offset data is likely to be located on the cache page.)
  - The last indirection to access the member is also very fast since it is a relative address to the instance itself —which hopefully is already in the cache.

`struct`/`class` Layout: Overhead (con't)

Using a **`virtual` member function** requires **at least one additional indirection** relative to a non-**`virtual`** function call:

One (1) indirection is needed to call the function if the function pointer is directly stored in the object.

- **Pro:** Fast since function pointer is extremely likely to be on the same L1 cache page as the instance.
- **Con:** RAM to store the **list** of function pointers **in each instance** plus the constructor overhead to set such **in each instance**.

NOTE: Each indirect function pointer call is slower than a direct call.

`struct`/`class` Layout: Overhead (con't)

Two (2) indirections are needed to call the function if the function pointer is stored in the vtable.

- **Pro:** Only one vtable pointer in each instance is required: constructors need only set one pointer.
- **Con:** An extra cache page and indirection:
  - The vtable is not likely to be close to the instance data and will need to use an additional cache page.
  - Slots can be likely to be located on the cached vtable page, so the slot (array) indirection will be negligible (only) for **direct `virtual`** function calls since the compiler already knows the slot offset at compile time. (If indirect then add another indirection.)
  - The last indirection is the indirect function call itself.

`struct`/`class` Layout: Overhead (con't)

Accessing a **`virtual` member function** through a **`virtual` member base** can require **three to four indirections** and likely at least an additional **cache page** access (for the vtable).

- Is your code also using pointers to everything as well? Hmmmm...
- There are some cases where the compiler can apply some optimizations —but, in general, this overhead is very real.

Indirections resulting in **cache misses** can be **costly** in terms of time.

- Try to design your algorithms to access **one type of object** at a time or in **small groups of the same types** to increase vtable cache hits.

Advice: Only use **`virtual`** to exploit true object-oriented, i.e., run-time, polymorphism.

Other Object-Oriented Languages

C++ is "special": Traditional and other object-oriented languages do **not** support **non-`virtual`** inheritance and **non-`virtual`** member functions.

When converting code from those languages to C++, do the following *before optimizing the code*:

- make all inheritance **`virtual`** and **`public`**,
- make all **abstract** functions **pure `virtual`**, and,
- make all non-**`static`** member functions **`virtual`**,
  - i.e., all **`static`** member functions remain **`static`**
  - i.e., make all object methods **`virtual`**
  - i.e., keep all **`static`** variables/methods **`static`**
  - i.e., keep all non-**`static`** variables/methods non-**`static`**
- represent **all uses of variables** that are **implicitly references** as **references**, or a suitable owning **smart pointer type**, e.g., `std::shared_ptr` or `std::unique_ptr`.

When To Use And Not To Use Inheritance

`public` **inheritance** should **only** be used to model **Is-A** or **Is-Completely-Substitutable-For-A** relationships for types.

Inheritance use needs to follow the Liskov Substitutability Principle (LSP):

Liskov Substitutability Principle [4]

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

When To Use And Not To Use Inheritance (con't)

Don't use **`public` inheritance** when: [6, 7]

- non-**`public`** inheritance is sufficient
- membership, i.e., **Has-A**, is sufficient
  - declaring as a member variable; encapsulation is nearly always sufficient
- the relationship is **Is-Almost-A**, e.g.,
  - A `Square` **cannot,** in practice, **inherit from** a `Rectangle` since a `Rectangle`'s equivalent of `set_width()` and `set_height()` members **will have adverse unacceptable side effects** when applied to a `Square`.
  - i.e., `Square` fails the LSP with respect to `Rectangle`.
- the relationship is **Is-Implemented-In-Terms-Of** since such is really a **Has-A** relationship.

When To Use And Not To Use Inheritance (con't)

Only use **`private` inheritance** to model **Has-A** relationships **if and only if**:

- one needs access to **`protected`** base class members, and/or,
- one needs to override a **`virtual`** function;

Otherwise use a member variable to model the **Has-A** relationship. [6, 7]

- Private inheritance **demotes** all **`public`** and **`protected`** base members to **`private`** access.

When To Use And Not To Use Inheritance (con't)

Only use **`protected` inheritance** to model **Is-A** relationships for derived types only and no access to everything else in the base classes.

In practice, there are few use cases where **`protected`** inheritance is useful.

- Protected inheritance **demotes** all **`public`** base members to **`protected`** access.

this

Given a user-defined **struct**, **class**, or **union** type, T, then in any constructor or non-**static** member function or in the destructor of T a **constant pointer** to the current instance of T, called **this**, is defined.

Internally this is passed to the constructor, destructor, or non-**static** member function as an **implicit argument**.

In most cases, **this**-> can be omitted in front of member names inside constructors, destructors, and non-**static** members.

- You will need to use **this**-> if the member is in a base class that is dependent on a template parameter.

this (con't)

Member functions can be adorned with cv-qualifiers, i.e., **const**, **volatile**, and **const volatile**, and/or with an Lvalue or Rvalue reference decorations.

- Such adornments adorn the type *T* that **this** points to within that member function and therefore **affect member access**.

w03/layout-egf1a.cxx

```
1  struct A {
2    void setValue(int);   // *this is "A"
3    int getValue() const; // *this is "A const"
4    int getValue() volatile; // *this is "A volatile"
5    int getValue() const volatile;
6      // *this is "A const volatile"
7    std::string getString() const&; // *this is "A const&"
8    std::string getString() &&; // *this is "A&&"
9    // etc.
10 };
```

Abstract Classes [3, [class.abstract]]

C++ has **no** special syntax to **denote** an **abstract class**.

An **abstract class** is a class that has **at least one pure `virtual` function** within it or *via* inheritance.

- A **pure `virtual` function** is called an **abstracti function** in other object-oriented languages.

An abstract class **can inherit** from a class that is **not** abstract.

By-value instances of abstract classes cannot be constructed —however references/pointers to abstract classes are allowed.

A base type's **pure `virtual`** function that has been **overridden** in a **non-pure manner** is no longer considered pure in the derived type.

Pure virtual Functions

C++ permits **abstract** member functions to be declared. These are called **pure virtual functions** and are written syntactically by prefixing the prototype with **virtual** and suffixing it with = 0:

w03/layout-egg1a.cxx

```
1  class abstract
2  {
3    // Always have a virtual destructor in an
4    // abstract/polymorphic class ...
5    virtual ~abstract() { }
6
7    // Pure virtual (abstract) function ...
8    virtual int move(int x, int y) = 0;
9  };
```

Pure `virtual` Functions (con't)

C++ permits **pure `virtual` functions** to be defined —although the function and type are still considered **pure** and the function must be **explicitly invoked** in a derived type if it is to be executed...

Pure virtual Functions (con't)

w03/layout-egg1b.cxx

```
1  class position
2  { // abstract class due to pure virtual functions
3  public:
4    virtual ~position() { } // ensure proper destruction
5    virtual int getX() const = 0;
6    virtual int getY() const = 0;
7    virtual void setX(int x) = 0;
8    virtual void setY(int y) = 0;
9    virtual void move(int dx, int dy) = 0;
10 };
11
12 void position::move(int dx, int dy) {
13   setX(getX()+dx); setY(getY()+dy);
14 }
```

Pure `virtual` Functions (con't)

w03/layout-egg1c.cxx

```
1  // euclidean does not override all pure virtual
2  // functions and so is still an abstract class ...
3  class euclidean : public position
4  {
5  public:
6    void move(int dx, int dy) override
7    {
8      position::move(dx, dy);
9    }
10 };
```

NOTE: The above example is a bit contrived. It is also very rare to write definitions for **pure `virtual`** functions. It works since the **pure `virtual`** requires a slot and the definition causes the function pointer to be set on that slot.

`struct` and `class`

There is no difference between **`struct`** or **`class`** in C++ *except* for their scope access default and some minor syntax issues/uses.

By default:

- **`struct`**s inherit **`public`**ly and members are declared **`public`**ly, and,
- **`class`**es inherit **`private`**ly and members are declared **`private`**ly.

Rule of thumb: Use **`struct`**s as one would in C and **`class`**es for OOP classes and virtually all other user-defined objects.

Member Initialization Lists

All versions of C++ have **member initialization lists**.

C++ **member initialization lists** are used in constructors to **invoke base class constructors** and to **construct** member variables.

If any base classes or members are **omitted** from the list, then they are **implicitly default constructed**.

The **member initialization list** appears **starting with a colon**, "**:**", **immediately after the close parenthesis**, "**)**", ending the constructor argument list.

It **ends with the opening brace**, "{", of the constructor definition's body.

The allowed syntax is that of **declaring comma-separated unnamed variables** to **invoke constructors** or and using the **member name** instead of a type to **invoke members' constructors**.

Member Initialization Lists (con't)

w03/mem-init-list1.cxx

```
1  struct base {
2    base() : b_(12) { }   // Construct b_ with 12.
3    int b_;
4  };
5  struct derived {
6    derived() :
7      base{}, d_(231)     // Default construct base; d_ = 231
8    { }
9    int d_;
10 };
11 void func(derived const&) { }
12 derived d;              // Default constructed
13 derived d2{};           // Default constructed
14 func(derived{});        // Unnamed and def. constructed.
15 // Next line is the "most vexing parse" ...
16 derived d3();           // This is a function pointer declaration.
```

Member Initialization Lists (con't)

A reason **member initialization lists** are needed is because once the open curly brace of the constructor code block is encountered **all base classes *and* all members have been successfully constructed**. Thus, the list allows the programmer to control the construction process of base classes and members.

Your code's **ordering of construction** should be as follows:

- **base classes** then
- **member variables** (in order of declared appearance in the class)

std::initializer_list

`std::initializer_list` is most commonly used with container and class types —but it can also be used with functions. [3, [support.initlist]]

It was add to C++11 to make it possible to **uniformly initialize all types** and to support a form of the C-style **brace-list initialization syntax**.

The `std::initializer_list` type is an container object with iterators defined in `<initializer_list>` and it allows the programmer to iterate through a **list of compile-time specified objects** as a result of using the uniform initialization brace syntax.

std::initializer_list (con't)

The list can be empty, or, it can be filled with literals/variables provided they are all of the same type. No implicit conversions or narrowings will occur. [3, [dcl.init.list]]

NOTE: If a type has a single-argument constructor with an `std::initializer_list` defined, then the `std::initializer_list` constructor will be preferred **if** the uniform initialization brace syntax is used to construct the type.

std::initializer_list (con't)

w03/init-list1.cxx

```cpp
#include <initializer_list>
#include <iostream>
#include <vector>

// Invokes std::vector<int>(std::initializer_list<int>)...
std::vector<int> v{ 1, 5, 23, -123, 45, 34 56 };
template <typename T>
T sum(std::initializer_list<T> l)
{
  T retval{};
  for (auto const& element : l)
    retval += element;
  return retval;
}
// ...
std::cout << sum({1, 2, 3}) << '\n'; // Outputs: 6
```

std::initializer_list (con't)

w03/init-list2.cxx

```
1  #include <initializer_list>
2  #include <list>
3
4  struct my_list
5  {
6    my_list(std::initializer_list<int> l) :
7      l_(l.begin(), l.end())
8      // i.e., Pass iterators to list<int> constructor
9    {
10   }
11
12   list<int> l_;
13 };
14
15 // Construct my_list with some ints ...
16 my_list wow{ 1, -53, 3, 123, 45, 234 };
```

Resource Acquisition Is Initialization (RAII)

Resource Acquisition Is Initialization is an extremely important **design pattern** in C++. It involves **writing objects to acquire and release resources** as follows:

- In the constructor(s), obtain the required resources.
- In the destructor, always free all obtained resources over the lifetime of the object.
- It is usually peferable to declare objects by value on the call stack to ensure the destructor is automatically called.

TIP: Instead of manually dynamically allocating memory and freeing it later, **write a class to manage that** in the constructor and the destructor.

- Even easier: this is what `std::unique_ptr` and `std::shared_ptr` are for!

Resource Acquisition Is Initialization (RAII) (con't)

In C++ when objects are **declared by value on the call stack their destructors are always called**:

- when they go out-of-scope normally, and,
- when they go out-of-scope when the stack is being unwound from a thrown exception.

Thus **cleaning up resources**, even in the presence of an exception, is made **simple** by this technique.

This technique is used extensively in C++ libraries. You should use it in your code as well.

Outline

Copying and MovingOverview

- C++98 supports object **copy** semantics.
- C++11 added object **move** semantics to the language.
- Moving an object is **potentially an optimized copy:**
  - $O(\text{moving data}) = O(\text{copying data})$
  - $\Omega(\text{moving data}) \neq 0$
  - Often, $\Omega(\text{moving data}) = \textit{cost}(\text{copying pointers to the data})$

What is Copying?

Copying an object A to another object B involves:

1. Destroy any data in B.
2. Ensure B is capable of holding A's data.
3. Copy all data in A into B.

ASIDE: If an exception occurs, B's state is invalid.

What is Copying? (con't)

Assuming `swap()` is **`noexcept`**, copying an object A to another object B in an **exception-safe** manner involves:

1. Declare a temporary, T, capable of holding A's data.
2. Copy all data in A into T.
3. If no exception occurred, `swap(B,T);`.
4. Destroy T.

NOTE: This is a **waste** of time and RAM if one **no longer needs / will destroy** A after the copy!

Copy and Move Elision

Copy/Move Elision

Should the cost of copying or moving ever appear to be zero, then **copy elision** was performed by the compiler **instead of** copying or moving objects multiple times. [1, §12.8 para. 31-32 [class.copy]]

Copy and Move Elision (con't)

From [2, §12.8 para. 31 [intro.memory]]:

- "When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the" constructor and destructor to be invoked "have side effects."
- When this is done, the extra invocations of constructor-destructor pairs are omitted (i.e., optimized away).

Benefit: Eliminates costly copying/moving of objects when returning from functions or throwing exceptions.

Copy Elision Example

To help programs fit better on slides, the following header file will be used:

w03/output.hxx

```
1 #ifndef OUTPUT_HXX_
2 #define OUTPUT_HXX_
3   #define OUT(S) std::cout << S << this << ' ' << '\n'
4   #define OUT2(S,A) std::cout << S << this << ' ' << A << '\n'
5 #endif // #ifndef OUTPUT_HXX_
```

Copy Elision Example (con't)

w03/copy-elision.cxx

```
#include <iostream>
#include "output.hxx"
class A {
  public:
    A() { OUT("A() "); }
    A(A const& b) { OUT2("A(copy) ",&b); }
    ~A() { OUT("~A() "); }
};

A a_function() {
  A a; // default construct A
  return a; // return copy of A
}

A value = a_function(); // Copy elision possible here.

int main() { }
```

Copy Elision Example (con't)

w03/copy-elision-output.txt

```
$ g++ -std=c++23 -Wall -Wextra -Werror
$ ./copy-elision.exe
A() 0×5581c5c6d151
~A() 0×5581c5c6d151
$
```

Copy/Move Elision Criteria

Elision can be performed by the compiler in these circumstances:

- in a function's **`return`** statement with a class return type when "the expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) with the same cv-unqualified type as the function return type"
- "when a temporary class object that has not been bound to a reference (12.2) would be copied/moved to a class object with the same cv-unqualified type"
- in certain *throw-expression*, and,
- in certain *exception-declaration*s of an exception handler.

[2, §12.8 para. 31-32 [intro.memory]]

Object Copy Semantics

C++98 supports user-defined object copy semantics by:

- defining a **copy constructor**, and,
- defining a **copy assignment operator**.

Object Copy Semantics (con't)

The copy constructor and copy assignment operator are both **required** to have a **single argument** that is a **const lvalue reference** to the object type.

e.g., Given the type Foo, the argument type must be Foo `const`&.

NOTE: Function arguments that are Foo `const`& in C++ accept both lvalue and rvalue Foo objects.

Copy Semantics Example 1

w03/copy.cxx

```
1  #include <iostream>
2  #include "output.hxx"
3  class A {
4    public:
5      A() { OUT("A() "); }
6      A(A const& b) { OUT2("A(copy) ",&b); }
7      ~A() { OUT("~A() "); }
8  };
9
10 A a_function() {
11   A a; // default construct A
12   return a; // return copy of A
13 }
14
15 A value = a_function(); // Copy elision possible here.
16
17 int main() { }
```

Copy Semantics Example 1 (con't)

w03/copy-output.txt

```
$ g++ -std=c++23 -Wall -Wextra -Werror
$ ./copy.exe
A() 0×55863d08c151
~A() 0×55863d08c151
$
```

Copy Semantics Example 2

w03/copy-large.cxx

```
1  #include <iostream>
2  #include <algorithm>
3  using namespace std;
4  class A
5  {
6    double *ptr;
7    public:
8      A() : ptr(new double[100]) { cout << "A() " << this << '\n'; }
9      A(A const& b) : ptr (new double[100]) {
10       cout << "A(copy) " << this << '\n';
11       std::copy(b.ptr, b.ptr+100, ptr);
12     }
13     A& operator =(A const& b) {
14       cout << "a=b;(copy) " << this << '\n';
15       std::copy(b.ptr, b.ptr+100, ptr);
16       return *this;
17     }
18     ~A() {
19       cout << "~A() " << this << '\n';
```

Copy Semantics Example 2 (con't)

```
20        delete[] ptr;
21      }
22      A operator +(A const& b) const {
23        cout << "a+b; " << this << '\n';
24        A retval(*this); // i.e., create local copy every invocation
25        for (int i{}; i != 100; ++i)
26          retval.ptr[i] += b.ptr[i];
27        return retval;
28      }
29  };
30
31  int main() {
32    A a, b, c;
33    c = a + b + b;
34  }
```

Copy Semantics Example 2 (con't)

w03/copy-large-output.txt

```
$ g++ -std=c++23 -Wall -Wextra -Werror
$ ./copy-large.exe
A() 0x7ffe8d8f8ba0
A() 0x7ffe8d8f8ba8
A() 0x7ffe8d8f8bb0
a+b; 0x7ffe8d8f8ba0
A(copy) 0x7ffe8d8f8bb8
a+b; 0x7ffe8d8f8bb8
A(copy) 0x7ffe8d8f8bc0
a=b;(copy) 0x7ffe8d8f8bb0
~A() 0x7ffe8d8f8bc0
~A() 0x7ffe8d8f8bb8
~A() 0x7ffe8d8f8bb0
~A() 0x7ffe8d8f8ba8
~A() 0x7ffe8d8f8ba0
$
```

Object Move Semantics

C++11 introduced the ability to move user-defined objects by:

- defining a **move constructor**, and,
- defining a **move assignment operator**.

Object Move Semantics (con't)

The move constructor and move assignment operator are both **required** to have a **single argument** that is an **(non-const) rvalue reference** to the object type.

e.g., Given the type Foo, the argument type must be Foo&&.

NOTE: Function arguments that are Foo&& in C++ can only accept rvalue Foo objects without any cv-qualifiers.

What is Moving?

Briefly:

- Copying A to B **copies the data** in A into B.
- Moving A to B **moves the data** in A into B.

What is Moving? (con't)

Clearly:

- All fundamental types (e.g., `int`, `double`) will always be copied.
- While pointer types (e.g., `int*`, `Foo*`) can only have their pointer values copied, what they point to does not necessarily need to be copied!

What is Moving? (con't)

References are conceptually equivalent to **`const`** pointers:

Pointer Declaration	Reference Declaration
`T*`	n/a
`T const*`	n/a
`T * const`	`T&`
`T const * const`	`T const&`

What is Moving? (con't)

The C++ standard states: "It is unspecified whether or not a reference requires storage (3.7)." [2, §8.3.2 para. 4 [idcl.ref]]

- e.g., the compiler might be able to perform additional optimizations when references are used —so avoid using the address of operator on a reference

What is Moving? (con't)

Notice the following:

Type Without a Name	Reference Type to That Type
T	T&&
T **const**	T **const**&&

What is Moving? (con't)

Reference types **referring to lvalues**:

- can be copied if `const` and needed copy operations are defined,
- can be moved if **cast** to an **rvalue** using `std::move()` or `std::forward<T>()` and needed move operations are defined.

What is Moving? (con't)

Reference types **referring to rvalues**:

- can be be moved if necessary move operations are defined,
- can be copied provided necessary copy operations are defined, or
  - Recall: In C++ all `const &` arguments of functions permit both `const` and non-`const` lvalues and rvalues to be passed to them.
  - This enables rvalues to be copied when needed move operations are not defined.
  - In effect, this means if something cannot be moved, then hopefully it can be copied without needing to write additional code to do such explicitly.

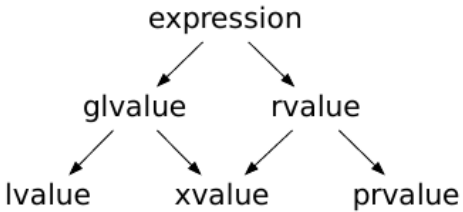
Lvalues and Rvalues

Excluding when explicit casts are used, in practice:

- an **lvalue** is a value/variable that **has a user-defined name**
  - `int i = 5;`
  - `int& lvalue_ref = i;`
  - `int&& rvalue_ref = 5;` is an lvalue that refers to an rvalue
- an **rvalue** is a value/variable that **does not have a user-defined name**
  - `5` is a literal value
  - `int(5)` is a variable with no name

Lvalues and Rvalues (con't)

More formally, C++ defines a taxonomy in [2, §3.10 Figure 1 [basic.lval]]:



Lvalues and Rvalues (con't)

From [2, §3.10 [basic.lval]]:

- "An *lvalue* [...] designates a function or an object."
- "An *xvalue* (an 'eXpiring' value) also refers to an object, usually near the of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references (8.3.2)."
- "An *glvalue* ('generalized' lvalue) is an lvalue or an xvalue."
- "An *rvalue* [...] is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object."
- "A *prvalue* ('pure' rvalue) is an rvalue that is not an xvalue."
  - e.g., the value of a literal such as `8.23e-1`

The Move "Algorithm"

Moving an object A into another object B involves:

1. If A isn't going to be immediately destroyed, or, if data leakage and/or security issues are a concern, then reset/zero/null out B's data.
2. `swap(A,B);`

NOTE: If this involves pointers to data structures, moving is much simpler than copying since it involves only copying and nulling-out pointers!

Move Semantics Example 1

w03/move.cxx

```
1  #include <iostream>
2  #include "output.hxx"
3  class A {
4  public:
5    A() { OUT("A() "); }
6    A(A const& b) { OUT2("A(copy) ",&b); }
7    A(A&& b) { OUT2("A(move) ",&b); }
8    A& operator =(A const& b) { OUT2("a=b;(copy) ",&b); return *this; }
9    A& operator =(A&& b) { OUT2("a=b;(move) ",&b); return *this; }
10   ~A() { OUT("~A() "); }
11 };
12 int main() {
13   A a, b; // default constructions
14   A c = A{}, d{std::move(a)}, e = std::move(b); // move constructions
15   a = b; // copy assignment
16   a = std::move(b); a = A{}; // move assignment
17 }
```

Move Semantics Example 1 (con't)

w03/move-output.txt

```
$ g++ -std=c++23 -Wall -Wextra -Werror
$ ./move.exe
A() 0×7ffd7164dd32
A() 0×7ffd7164dd33
A() 0×7ffd7164dd34
A(move) 0×7ffd7164dd35 0×7ffd7164dd32
A(move) 0×7ffd7164dd36 0×7ffd7164dd33
a=b;(copy) 0×7ffd7164dd32 0×7ffd7164dd33
a=b;(move) 0×7ffd7164dd32 0×7ffd7164dd33
A() 0×7ffd7164dd37
a=b;(move) 0×7ffd7164dd32 0×7ffd7164dd37
~A() 0×7ffd7164dd37
~A() 0×7ffd7164dd36
~A() 0×7ffd7164dd35
~A() 0×7ffd7164dd34
~A() 0×7ffd7164dd33
~A() 0×7ffd7164dd32
$
```

Move Semantics Example 2

In this example:

- `class A` stores a `double` array of length size.
- If `operator +(A,A)` is invoked the sum of the two arrays is computed.
- If `A(length)` is constructed, the array size is set to length with all initial values set to zero.
- If `A(A,length)` is constructed, the array is copied and length is set to the maximum of the array size or length.
- Note the use of lvalues and rvalues with `operator +()`.

Move Semantics Example 2 (con't)

w03/move-large.cxx

```
1  #include <iostream>
2  #include <algorithm>
3  #include <utility>
4  #include <initializer_list>
5  class A
6  {
7    std::size_t size_;
8    double *ptr_;
9
10   void size_adjust(A const& b)
11   {
12     if (size_ < b.size_)
13     { // Extend *this to be as large as b...
14       std::cout << "size_adjust() " << this << '\n';
15       A tmp{b, b.size_}; swap(tmp);
16     }
17     // else *this is same or larger than b...
18   }
19
20 public:
21   void swap(A& b) noexcept
22   {
23     std::swap(ptr_, b.ptr_);
24     std::swap(size_, b.size_);
25   }
26
```

Move Semantics Example 2 (con't)

```
27      std::size_t size() const noexcept
28      {
29        return size_;
30      }
31
32      double& operator [](std::size_t i) const noexcept
33      {
34        return ptr_[i];
35      }
36
37      A() :
38        size_{},
39        ptr_(nullptr)
40      {
41        std::cout << "A() " << this << '\n';
42      }
43
44      A(std::size_t sz) :
45        size_{sz},
46        ptr_{new double[sz]}
47      {
48        std::cout << "A(" << sz << ") " << this << '\n';
49      }
50
51      ~A()
52      {
53        std::cout << "~A() " << this << '\n';
```

Move Semantics Example 2 (con't)

```
54        delete[] ptr_;
55      }
56
57      A(std::initializer_list<double> il) :
58        size_{il.size()},
59        ptr_{new double[il.size()]}
60      {
61        std::cout << "A(init_list:" << size_ << ") " << this << '\n';
62        std::copy(il.begin(), il.end(), ptr_);
63      }
64
65      A(A const& b, std::size_t sz) :
66        size_{std::max(b.size_,sz)},
67        ptr_(new double[size_])
68      {
69        std::cout << "A(copy," << sz << ") " << this << ' ' << &b << '\n';
70
71        std::copy(b.ptr_, b.ptr_+b.size_, ptr_); // copy data
72        std::fill_n(ptr_, size_-b.size_, double{}); // zero data
73      }
74
75      A(A const& b) :
76        size_{b.size_}, ptr_(new double[b.size_])
77      {
78        std::cout << "A(copy) " << this << ' ' << &b << '\n';
79        std::copy(b.ptr_, b.ptr_+b.size_, ptr_); // copy data
80      }
```

Move Semantics Example 2 (con't)

```
81
82      A& operator =(A const& b) {
83        std::cout << "a=b;(copy) " << this << ' ' << &b << '\n';
84        if (size_ != b.size_) {
85          A tmp{b}; swap(tmp); // copy and swap
86        } else {
87          size_adjust(b); // adjust size if needed
88          std::copy(b.ptr_, b.ptr_+b.size_, ptr_); // copy data
89          size_ = b.size_;
90        }
91        return *this;
92      }
93
94      A(A&& b) :
95        size_{std::move(b.size_)}, // move size
96        ptr_{std::move(b.ptr_)} // move pointer
97      {
98        std::cout << "A(move) " << this << ' ' << &b << '\n';
99
100       b.size_ = {}; // zero out original size
101       b.ptr_ = nullptr; // null out original pointer
102     }
103
104     A& operator =(A&& b)
105     {
106       std::cout << "a=b;(move) " << this << ' ' << &b << '\n';
107       swap(b); // swap
```

Move Semantics Example 2 (con't)

```
108       return *this;
109     }
110
111     A operator +(A const& b) const
112     {
113       std::cout << "a+b;(const) " << this << ' ' << &b << '\n';
114
115       A retval{*this, std::max(size_, b.size_)}; // copy *this
116       for (std::size_t i{}; i != b.size_; ++i)
117         retval.ptr_[i] += b.ptr_[i];
118       return retval;
119     }
120
121     A operator +(A&& b) const {
122       std::cout << "a+b;(rvalue1) " << this << ' ' << &b << '\n';
123       b.size_adjust(*this); // adjust size if needed
124
125       for (std::size_t i{}; i != size_; ++i)
126         b.ptr_[i] += ptr_[i]; // Add *this.ptr to b.ptr!
127       return std::move(b); // Move b into return value
128     }
129   };
130
131   A operator +(A&& a, A const& b) {
132     std::cout << "a+b;(rvalue2) " << &a << ' ' << &b << '\n';
133     return b+std::move(a);
134   }
```

Move Semantics Example 2 (con't)

```
135
136  A operator +(A&& a, A&& b) {
137    std::cout << "a+b;(rvalue3) " << &a << ' ' << &b << '\n';
138    return a+std::move(b);
139  }
140
141  int main() {
142    A result = A(5) + A{0.0, 1.1, 2.2, 3.3, 4.4};
143  }
```

Move Semantics Example 2 (con't)

w03/move-large-output.txt

```
$ g++ -std=c++23 -Wall -Wextra -Werror
$ ./move-large.exe
A(init_list:5) 0×7ffe53f0f340
A(5) 0×7ffe53f0f330
a+b;(rvalue3) 0×7ffe53f0f330 0×7ffe53f0f340
a+b;(rvalue1) 0×7ffe53f0f330 0×7ffe53f0f340
A(move) 0×7ffe53f0f320 0×7ffe53f0f340
~A() 0×7ffe53f0f330
~A() 0×7ffe53f0f340
~A() 0×7ffe53f0f320
$
```

When To Use std::move() and std::forward<T>()

Use `std::move()` when non-**`const`** lvalue needs to be moved.

Use `std::forward<T>(value)` when `value`'s type can be an lvalue or rvalue reference when `value`'s is a template parameter.

`std::forward<T>(value)` when used with templates and the special function argument pattern `T&&` where `T` is a template parameter enables the **perfect forwarding** of function arguments.

std::forward<T>() Example

w03/std-forward.cxx

```
1  #include <cmath>
2  #include <iostream>
3  #include "output.hxx"
4  using namespace std;
5
6  struct Foo
7  {
8    double d_;
9
10   // Permit initialization of Foo instance with a double...
11   explicit Foo(double d) : d_{d} {
12     cout << "Foo(" << d_ << ") " << this << '\n';
13   }
14
15   // Permit implicit cast to double...
16   operator double() const { return d_; }
17   // Constructors, assignment operators, and destructors w/outputs...
18   Foo() : d_{} { OUT("Foo() "); }
19   Foo(Foo const& f) : d_{f.d_} { OUT2("Foo(copy) ",&f); }
```

std::forward<T>() Example (con't)

```
20      Foo(Foo&& f) : d_{f.d_} { OUT2("Foo(move) ",&f); }
21      Foo& operator =(Foo const& f) {
22        OUT2("f=g;(copy) ",&f);
23        d_ = f.d_; return *this;
24      }
25      Foo& operator =(Foo&& f) {
26        OUT2("f=g;(move) ",&f);
27        d_ = f.d_; return *this;
28      }
29      ~Foo() { OUT("~Foo() "); }
30    };
31    template <typename Op, typename Arg>
32    constexpr auto simple_proxy(Op&& op, Arg&& arg) {
33      return op(std::forward<Arg>(arg));
34    }
35    template <typename Op, typename... Args>
36    constexpr auto fancy_proxy(Op&& op, Args&&... args) {
37      return op(std::forward<Args>(args)...);
38    }
39    constexpr double my_func(double a, double b) { return a+b; }
```

std::forward<T>() Example (con't)

```
40
41  int main() {
42    cout << "sp:\n";
43    cout << simple_proxy<double(double)>(std::sin, Foo{0.0}) << "\n";
44    cout << "\nfp:\n";
45    cout << fancy_proxy(my_func, Foo{1.1}, Foo{2.2}) << '\n';
46  }
```

std::forward<T>() Example (con't)

w03/std-forward-output.txt

```
$ g++ -std=c++23 -Wall -Wextra -Werror
$ ./std-forward.exe
sp:
Foo(0) 0x7ffff6dfb6e0
0
~Foo() 0x7ffff6dfb6e0

fp:
Foo(2.2) 0x7ffff6dfb6e0
Foo(1.1) 0x7ffff6dfb6d8
3.3
~Foo() 0x7ffff6dfb6d8
~Foo() 0x7ffff6dfb6e0
$
```

References

[1] ISO/IEC. *Information technology – Programming languages – C++*. ISO/IEC 14882-2011, IDT. Geneva, Switzerland, 2012.

[2] ISO/IEC. *Information technology – Programming languages – C++*. ISO/IEC 14882-2014, IDT. Geneva, Switzerland, 2014.

[3] ISO/IEC. *Information technology – Programming languages – C++*. ISO/IEC 14882-2024, IDT. Geneva, Switzerland, 2024.

[4] Barbara Liskov. "Data Abstraction and Hierarchy". In: *SIGPLAN Notices* 23 (5 May 1988).

[5] Bjarne Stroustrup. *Design and Evolution of C++*. Boston, MA: Addison-Wesley, 1994. ISBN: 978-0-201-54330-8.

[6] Herb Sutter. *Exceptional C++*. Boston, MA: Addison-Wesley, 2000. ISBN: 978-0-201-61562-3.

References (con't)

[7] Herb Sutter. *More Exceptional C++*. Boston, MA: Addison-Wesley, 2002. ISBN: 978-0-201-70434-1.