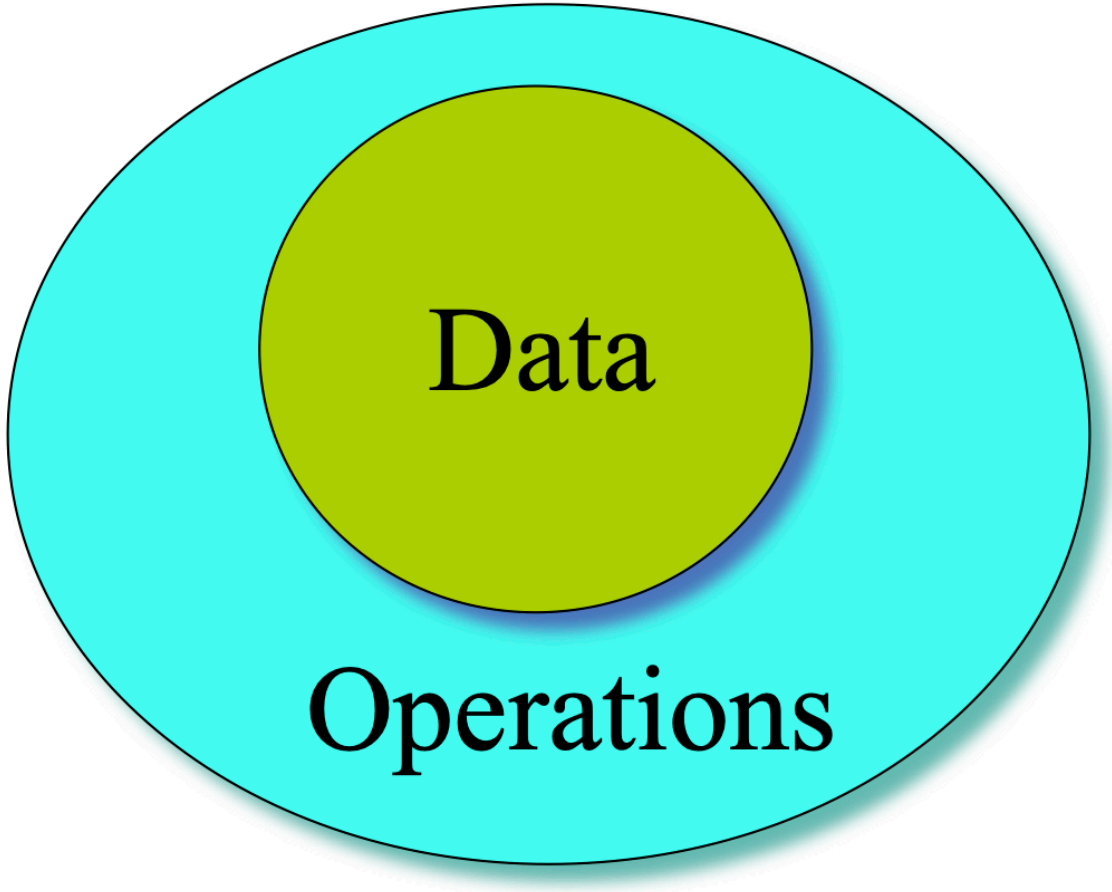
Inside Object

- Access control
- Objects in different places
- Static
- Reference
- Const
- Operator `new` and `delete`

Object = Attributes + Services

- Data: the properties or status
- Operations: the functions



Access control

- The members of a class can be cataloged, marked as:
  - `public`
  - `private`
  - `protected`

`public`

- public means all member declarations that follow are available to everyone.

Example: Public.cpp

`private`

- The private keyword means that no one can access that member except inside function members of that type.

Example: Private.cpp

Friend

- is a way to explicitly grant access to a function that isn't a member of the structure
- The class itself controls which code has access to its members.
- Can declare a global function as a friend, as well as a member function of another class, or even an entire class, as a friend.

Example: Friend.cpp

class vs. struct

- class defaults to private
- struct defaults to public

Example: Class.cpp

Where are the objects?

Local object

- Local variables are defined inside a method, have a scope limited to the method to which they belong.

```
int TicketMachine::refundBalance() {
    int amountToRefund;
    amountToRefund = balance;
    balance = 0;
    return amountToRefund;
}
```

- A local variable of the same name as a field will prevent the field being accessed from within a method.

Fields, parameters, local variables

- All three kinds of variable are able to store a value that is appropriate to their defined type.
- Fields are defined outside constructors and methods
- Fields are used to store data that persists throughout the life of an object. As such, they maintain the current state of an object.They have a lifetime that lasts as long as their object lasts.
- Fields have class scope: their accessibility extends throughout the whole class, and so they can be used within any of the constructors or methods of the class in which they are defined.

- As long as they are defined as private, fields cannot be accessed from anywhere outside their defining class.
- Formal parameters and local variables persist only for the period that a constructor or method executes.Their lifetime is only as long as a single call, so their values are lost between calls.As such, they act as temporary rather than permanent storage locations.
- Formal parameters are defined in the header of a constructor or method.They receive their values from outside, being initialized by the actual parameter values that form part of the constructor or method call.

- Formal parameters have a scope that is limited to their defining constructor or method.
- Local variables are defined inside the body of a constructor or method.They can be initialized and used only within the body of their defining constructor or method.
- Local variables must be initialized before they are used in an expression – they are not given a default value.
- Local variables have a scope that is limited to the block in which they are defined.They are not accessible from anywhere outside that block.

Global objects

- Consider

```
#include "X.h"
X global_x(12, 34);
X global_x2(8, 16);
```

- Constructors are called before `main()` is entered
  - Order controlled by appearance in file
  - In this case, `global_x` before `global_x2`
  - `main()` is no longer the first function called
- Destructors called when
  - `main()` exits, or
  - `exit()` is called

Static Initialization Dependency

- Order of construction within a file is known
- Order between files is unspecified!
- Problem when non-local static objects in different files have dependencies.
- A non-local static object is:
  - defined at global or namespace scope
  - declared static in a class
  - defined static at file scope

Static Initialization Solutions

- Just say no -- avoid non-local static dependencies.
- Put static object definitions in a single file in correct order.

Static

- Two basic meanings
  - Static storage
  - Restricted access
- allocated once at a fixed address
  - Visibility of a name
  - internal linkage
- In C++, don't use static except inside functions and classes
  - Don't use `static` to restrict access

Uses of “static” in C++

type	meaning
static free function	internal linkage(deprecated)
static global variables	internal linkage(deprecated)
static local variables	persistent storage
static member variables	shared by instances
static member functions	shared by instances, can only access static members

Global static hidden in file

File1

```
int g_global;
static int s_local;

void
func() {
  ...
}

static
void
hidden() { ...}
```

File2

```
extern int g_global;
void func();

extern int s_local;
int
myfunc() {
  g_global += 2;
  s_local *= g_global;
  func();
}
```

?

Static inside functions

- Value is remembered for entire program
- Initialization occurs only once
- Example:
    - count the number of times the function has been called

```
void f() {
   static int num_calls = 0;
   //...
   num_calls++;
}
```

Static applied to objects

- Suppose you have a class

```
class X {
   X(int, int);
   ~X();
   // ...
};
```

- And a function with a static X object

```
void f() {
   static X my_X(10, 20);
   ...
}
```

Static applied to objects ...

- Construction occurs when definition is encountered
  - Constructor called at-most once
  - The constructor arguments must be satisfied
- Destruction takes place on exit from program
  - Compiler assures LIFO order of destructors

Conditional construction

- Example: conditional construction

```
void f(int x) {
   if (x > 10) {
      static X my_X(x, x * 21);
      // ...
}
```

- `my_X`
  - is constructed once, if `f()` is ever called with `x > 10`
  - retains its value
- destroyed only if constructed

Can we apply static to members?

- Static means
  - Hidden
  - Persistent
- Hidden: *A static member is a member*
  - Obeys usual access rules
- Persistent: Independent of instances
- Static members are class-wide
  - variables, or
  - functions

Static members

- Static member variables
  - Global to all class member functions
  - Initialized once, at file scope
  - provide a place for this variable and init it in .cpp
  - No `static` in .cpp

Example: StatMem.h, StatMem.cpp

Static members

- Static member functions
  - Have no implicit receiver (`this`)
    - (why?)
  - Can access only static member variables
    - (or other globals)
  - No `static` in .cpp

    – Can’t be dynamically overridden

Example: StatFun.h, StatFun.cpp

To use static members

```
<class name>::<static member>
<object variable>.<static member>
```

Reference

Declaring references

- Reference is a new way to manipulate objects in C++

```
char c;         // a character
char *p = &c; // a pointer to a character
char &r = c;  // a reference to a character
```

- Local or global variables
  - `type& refname = name;`
  - For ordinary variables, the initial value is required
- In parameter lists and member variables
  - `type& refname`
  - Binding defined by caller or constructor

References

- Declares a new name for an existing object

```
int X=47;
int &Y=X; // Y is a reference to X
```

// X and Y now refer to the same variable

cout<<"Y="<<y; //printsY=47

Y = 18;

cout<<"X="<<x; //printsX=18

Rules of references

- References must be initialized when defined
- Initialization establishes a binding
- In declaration

```
int x = 3;
int &y = x;
const int &z = x;
```

- As a function argument

```
void f ( int& x );
f(y); // initialized when function is called
```

Rules of references(cn'td)

- Bindings don’t change at run time, unlike pointers

  Assignment changes the object referred-to

```
int &y = x;
y = 12; // Changes value of x
```

- The target of a reference must have a location!

```
void func(int &);
func (i * 3);       // Warning or error!
```

Pointers vs. References

- References
  - can't be null
  - are dependent on an existing variable, they are an alias for an variable
  - can't change to a new "address" location
- Pointers
  - can be set to null
  - pointer is independent of existing objects
  - can change to point to a different address

Restrictions

- No references to references
- No pointers to references

```
int &*p;        // illegal
```

- Reference to pointer is ok

```
void f(int *&p);
```

- No arrays of references

Left Value vs Right Value

- Left-value can be simply regarded as value that can be used at the left of assignment:
  - Variable, reference
  - Result of operator `*`, `[]`, `.` and `->`
- Right-value are values can be used at the right hand of the assignment:
  - Literal
  - Expression
- A reference parameter can take left-value only —> reference is the alias of a left-value
- The passing of an argument is initializing of the parameter

Right-value reference

```
int x=20; // left-value
int&& rx = x * 2;
// the result of x*2 is a right-value, rx extends its lifttime
int y = rx + 2; // In this way it can be reused:42
rx = 100;
// Once a right-value reference is initialized,
// this variable becomes a left-value that can be assigned
int&& rrx1 = x;
// Illegal: right-value reference can not be initialed by a left-value
const int&& rrx2 = x;
// Illegal: right-value reference can not be initialed by a left-value
```

Parameter as right-value

```
// take left-value
void fun(int& lref) {
    cout << "l-value" << endl;
}
// take right-value
void fun(int&& rref) {
    cout << "r-value" << endl;
}

int main() {
    int x = 10;
    fun(x); // output: l-value reference
    fun(10); // output: r-value reference
}
```

Const reference as parameter

```
void fun(const int& clref) {
   cout << "l-value const reference\n";
}
```

- Such a function can accept right-value when there is no function takes right-vaue

Constant

Const

- declares a variable to have a constant value

```
const int x = 123;
x = 27; // illegal!
x++; // illegal!
int y = x; // Ok, copy const to non-const
y = x;     // Ok, same thing
const int z = y; // ok, const is safer
```

Constants

- Constants are variables
  - Observe scoping rules
  - Declared with `const` type modifier
- A const in C++ defaults to internal linkage
  - the compiler tries to avoid creating storage for a const -- holds the value in its symbol table.
  - extern forces storage to be allocated.

Compile time constants

```
const int bufsize = 1024;
```

- value must be initialized
- unless you make an explicit extern declaration:

```
extern const int bufsize;
```

- Compiler won't let you change it

Run-time constants

- const value can be exploited

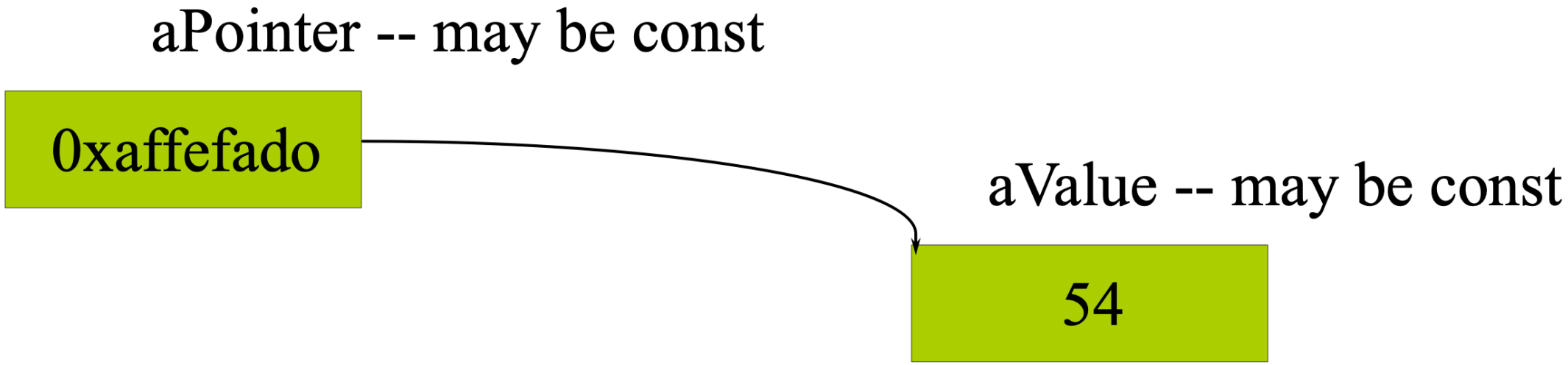
```
const int class_size = 12;
int finalGrade[class_size]; // ok
int x;
cin >> x;
const int size = x;
double classAverage[size]; // error!
```

Aggregates

- It’s possible to use const for aggregates, but storage will be allocated. In these situations, const means “a piece of storage that cannot be changed.”
- However, the value cannot be used at compile time because the compiler is not required to know the contents of the storage at compile time.

```
const int i[] = { 1, 2, 3, 4 };
float f[i[3]]; // Illegal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
double d[s[1].j]; // Illegal
```

Pointers and const




```
char * const q = "abc"; // q is const
*q = 'c'; // OK
q++; // ERROR
const char *p = "ABCD"; // (*p) is a const char
*p = 'b'; // ERROR! (*p) is the const
```

Try: What do these mean?

```
string p1("Fred");
const string* p = &p1;
string const* p = &p1;
string *const p = &p1;
```

Pointers and constants

	`int i;`	`const int ci = 3;`
`int *ip;`	`ip = &i;`	`ip = &ci; // ERROR`
`const int *cip`	`cip = &i;`	`cip = &ci;`

- Remember:

```
*ip  = 54;  // always legal since ip points to int
*cip = 54;  // never legal since cip points to const int
```

String Literals

```
char* s = "Hello, world!";
```

- `s` is a pointer initialized to point to a string constant
- This is actually a `const char *s` but compiler accepts it without the `const`
- Don't try and change the character values (it is undefined behavior)
- If you want to change the string, put it in an array:

```
char s[] = "Hello, world!";
```

Conversions

- Can always treat a non-const value as const

```
void f(const int* x);
int a = 15;
f(&a); // ok
const int b = a;

f(&b); // ok
b = a + 1; // Error!
```

You cannot treat a constant object as non-constant without an explicit cast (`const_cast`)

Passing by const value?

```
void f1(const int i) {
    i++; // Illegal -- compile-time error
}
```

Returning by const value?

```
int f3() { return 1; }
const int f4() { return 1; }
int main() {
    const int j = f3(); // Works fine
    int k = f4(); // But this works fine too!
}
```

Passing and returning addresses

- Passing a whole object may cost you a lot. It is better to pass by a pointer. But it's possible for the programmer to take it and modify the original value.
- In fact, whenever you're passing an address into a function, you should make it a `const` if at all possible.

Example: ConstPointer.cpp, ConstReturning.cpp

const object

Constant objects

- What if an object is const?

```
const Currency the_raise(42, 38);
```

- What members can access the internals?
- How can the object be protected from change?
- Solution: declare member functions const
    - Programmer declares member functions to be safe

Const member functions

- Cannot modify their objects

```
int Date::set_day(int d){
    //...error check d here...
    day = d;    // ok, non-const so can modify
}
int Date::get_day() const {
    day++;         //ERROR modifies data member
    set_day(12); // ERROR calls non-const member
    return day;  // ok
}
```

Const member function definition

- Repeat the const keyword in the definition as well as the declaration

```
int get_day () const;
int get_day() const { return day };
```

- Function members that do not modify data should be const member functions are safe for const objects declared const

Const objects

- Const and non-const objects

```
// non-const object
Date  when(1,1,2001);      // not a const
int day = when.get_day(); // OK
when.set_day(13);          // OK
// const object
const Date birthday(12,25,1994);  // const
int day = birthday.get_day();      // OK
birthday.set_day(14);            // ERROR
```

Overloaded `const` and none-const functions

```
void f() const;
void f();
```

Constant fields in class

```
class A {
   const int i;
};
```

- has to be initialized in initializer list of the constructor

Compile-time constants in classes

```
class HasArray {
   const int size;
   int array[size]; // ERROR!
};
```

- use "anonymous enum" hack

```
class HasArray {
   enum { size = 100 };
   int array[size]; // OK!
};
```

- Or make the const value static:

```
class HasArray {
   static const int size = 100;
   int array[size];
}
```

- static indicates only one per class (not one per object)

Dynamically allocated memory

- `new`
  - `new int;`
  - `new Stash;`
  - `new int[10]`
- `delete`
  - `delete p;`
  - `delete[] p;`
- new is the way to allocate memory as a program runs. Pointers become the only access to that memory
- delete enables you to return memory to the memory pool when you are finished with it.
- `{}` can be used to pass init values to object(s) `new` generted.

Dynamic Arrays

```
int * psome = new int [10];
delete[] psome;
```

- The `new` operator returns the address of the first element of the block.
- The presence of the brackets tells the program that it should free the whole array, not just the element

The new-delete mech.

```
int *p=new int;
int *a=new int[10];
Student *q=new Student();
Student *r=new Student[10];
delete p;
a++;
delete[] a;
delete q;
delete r;
delete[] r;
```

Tips for new and delete

- Don't use `delete` to free memory that `new` didn't allocate.
- Don't use `delete` to free the same block of memory twice in succession.
- Use `delete []` if `new []` was used to allocate an array.
- Use `delete` (no brackets) if `new` was used to allocate a single entity.
- It's safe to apply `delete` to the null pointer (nothing happens).

What we've learned today?

- Access control
- Objects in different places
- Static
- Reference
- Const
- Operator `new` and `delete`