Creative Software Design

Modern C++

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Topics Covered

- History of C++
- Various useful features in modern C++
- Smart pointers

History of C++

- C++ is a continuously evolving language
  - Most recent C++ version (C++23) is published in 2024 and the next version (C++26) is being developed

- Before C++ standard
  - In 1979, Bjarne Stroustrup at Bell Labs developed C++ as an extension of the C language
  - In 1990 and 1991, ANSI and ISO C++ standard committee founded, respectively

- Traditional C++
  - In 1998, the first C++ standard, C++98 was published
    - (Almost) all features in C++ learned so far is included in C++98
  - In 2003, C++03, minor update of C++98 was published
    - C++03 does not introduce a new feature, but make corrections to C++98

History of C++

- Until 2011, there was no significant update to C++
  - It means that there was no update to C++ from 2003 to 2011

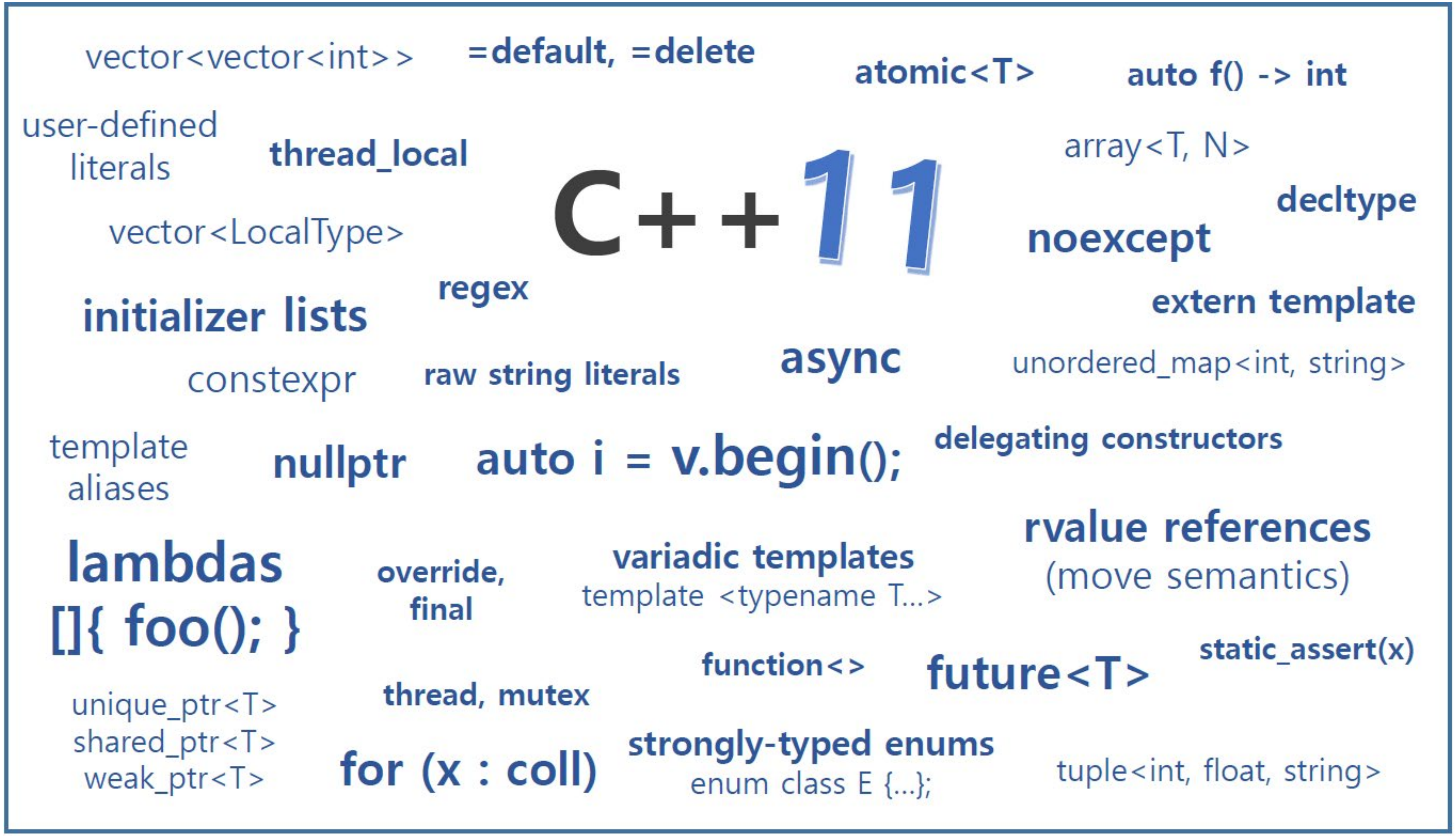
- Modern C++
  - C++11, C++14, , C++17, C++20, and C++23
  - C++ standard is published every 3 years

C++ is Really Evolving

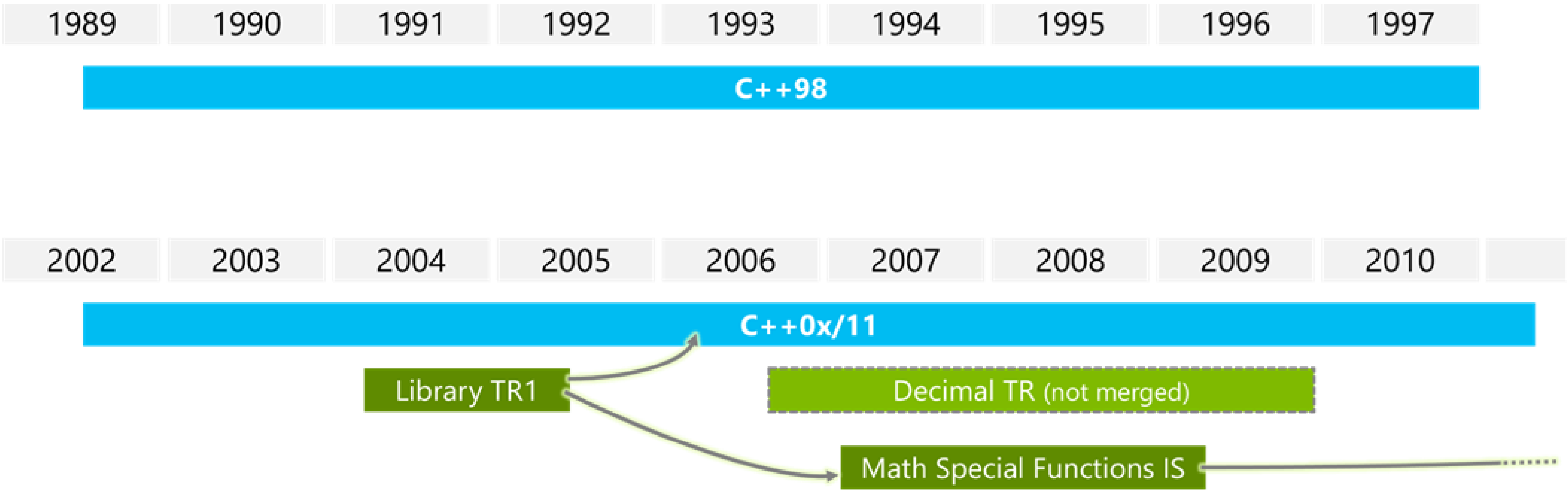
- Modern C++ includes a lot futures and becomes complex more and more

C++ version	#pages in standard documents	delta in #pages	
C++98	732		
C++03	757	25	(3.4%)
C++11	1338	581	(76.8%)
C++14	1358	20	(1.5%)
C++17	1605	247	(18.2%)
C++20	1853	248	(15.5%)
C++23	2104	251	(13.5%)

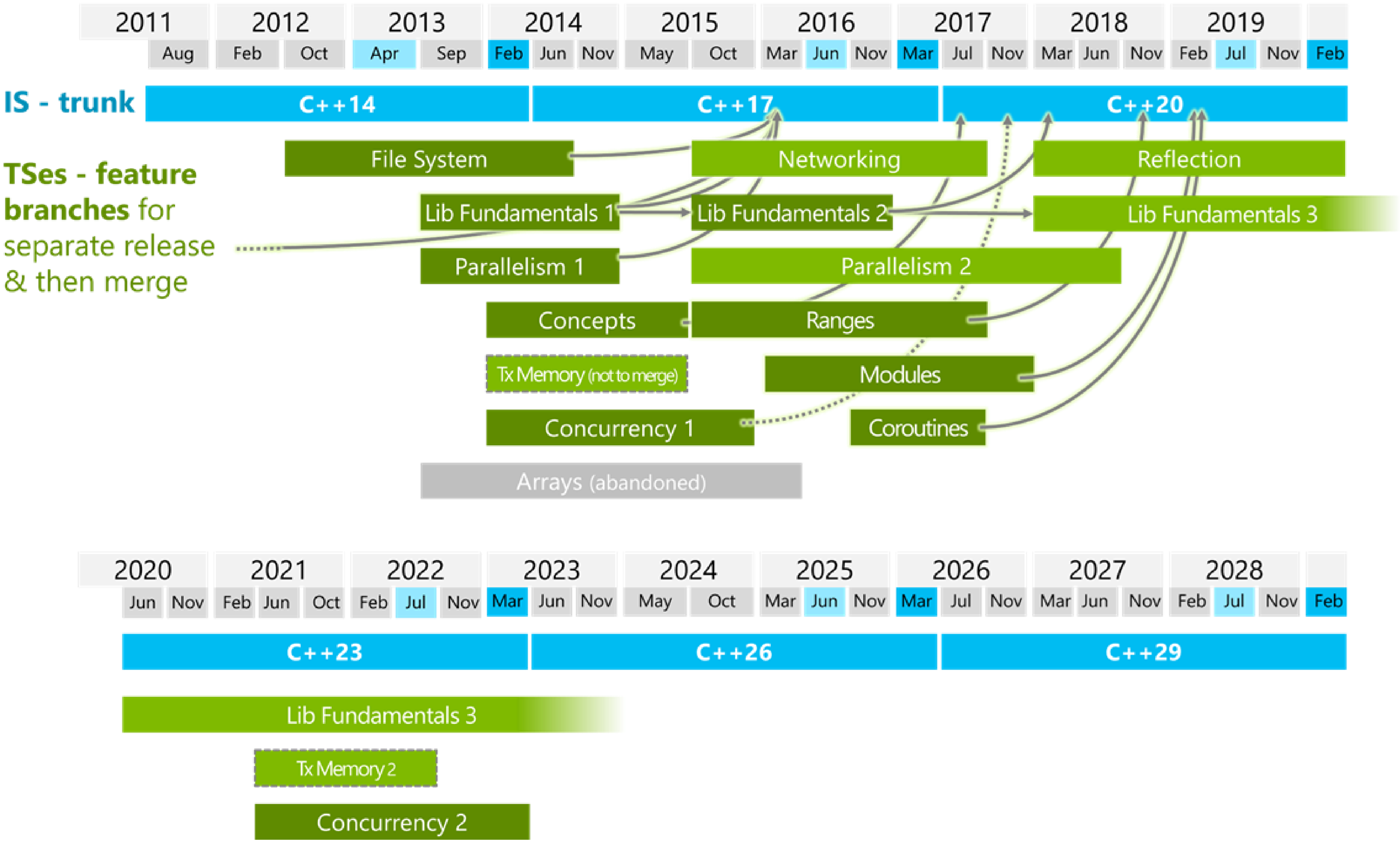
New features in C++11 (and C++14)



Old Days of C++


1989
1990
1991
1992
1993
1994
1995
1996
1997
C++98
2002
2003
2004
2005
2006
2007
2008
2009
2010
C++0x/11
Library TR1
Decimal TR (not merged)
Math Special Functions IS

Evolving C++



2011 2012 2013 2014 2015 2016 2017 2018 2019
Aug Feb Oct Apr Sep Feb Jun Nov May Oct Mar Jun Nov Mar Jul Nov Mar Jun Nov Feb Jul Nov Feb
IS - trunk
C++14
C++17
C++20
TSes - feature branches for separate release & then merge
File System
Networking
Reflection
Lib Fundamentals 1
Lib Fundamentals 2
Lib Fundamentals 3
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (not to merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)
2020 2021 2022 2023 2024 2025 2026 2027 2028
Jun Nov Feb Jun Oct Feb Jul Nov Mar Jun Nov May Oct Mar Jun Nov Mar Jul Nov Mar Jun Nov Feb Jul Nov Feb
C++23
C++26
C++29
Lib Fundamentals 3
Tx Memory 2
Concurrency 2

auto

- `auto` keywords specifies the type of the variable that is being declared automatically
  - The actual type will be deduced from its initializer

```
map<string, string>::const_iterator it = m.cbegin();
double const param = config["param"];
singleton& s = singleton::instance();
```

```
auto it = m.begin();
auto const param = config["param"];
auto& s = singleton::instance();
```

auto

- `auto` cannot be used for function and template type parameters in C++11
  - It's because parameter type cannot be deduced at compile-time

```
void f(auto a, auto b){...} // not allowed
```

- Since C++17, you can use auto in the template type parameters

```
template<auto n>
struct B { /* ... */ };
B<5> b1;   // OK: non-type template parameter type is int
```

- Since C++20, you can use `auto` in the parameter list of a function

```
void f1(auto); // same as template<class T> void f1(T)
```

decltype

- Suppose that you are writing a function template that takes two parameters and return the sum of them

```
template<class T, class U>
??? add(T x, U y)
{
  return x + y;
}
```

- What should be the return type of add()?
  - T? U? something else?
  - Can we know the type of return at compile-time?

decltype

- Can we know the type of return at compile-time?
  - Yes we can. Use `decltype`
- `decltype(expr)` represents the type of an expr

```
int a;
decltype(a) b; // same as int b;
```

```
template<class T, class U>
auto add(T x, U y) -> decltype(x+y)
{
  return x + y;
}
```

- The return type of add() is the type of (x+y)

auto as a Return Type

- Since C++14, you can use auto in the declared return type of a function
  - We call it return type deduction

```
auto g() { return 0.0; } // OK since C++14: g returns double
```

- Return type deduction is not always possible.

```
auto f(bool val)
{
    if (val) return 123; // deduces return type int
    else return 3.14f;   // Error: deduces return type float
}
```

- FYI: https://en.cppreference.com/w/cpp/language/function#Return_type_deduction

auto and decltype

- `auto` and `decltype` can be used together

```
int x = 1;
// return type is int, same as decltype(x)
decltype(auto) f() { return x; }
// return type is int&, same as decltype((x))
decltype(auto) f() { return(x); }
```

Range-based for

- Recall to for statements in Python
  - How can we iterate each element of a list?

```
mylist = [1,2,3]
for i in mylist:
  ...
```

- How can we do the (almost) same thing in C++?

```
vector<int> mylist = {1,2,3};
for (vector<int>::iterator it = mylist.begin(); it != mylist.end(); ++it)
{...}
```

or

```
vector<int> mylist = {1,2,3};
for (auto it = mylist.begin(); it != mylist.end(); ++it)
{...}
```

Range-based for

- It's still too long!
  - Why do we need to repeat typing XX.begin(), XX.end(), etc?

```
vector<int> mylist = {1,2,3};
for (auto it = mylist.begin(); it != mylist.end(); ++it)
{...}
```

- C++11 introduces range-based for

```
vector<int> mylist = {1,2,3};
for (auto& it : mylist){ ... }
```

```
mylist = [1,2,3]
for i in mylist:
  ...
```

override

- One of the common mistake on C++
  - `MyDerived::f1(int f)` does not overrides `MyBase::f1(float f)` because their parameter type is different

```
class MyBase{
public:
    virtual void f1(float f){
        cout << "MyBase::f1(float f)" << endl;
    }
};
class MyDerived :public MyBase{
public:
    virtual void f1(int f){
        cout << "MyDerived::f1(int f)" << endl;
    }
};
int main(){
    MyBase *pd = new MyDerived();
    pd->f1(2.4); // what will be printed?
    return 0;
}
```

override

- How can we avoid such mistake?
  - Explicitly specify that `MyDerived::f1()` should override some function in its super class

```
class MyBase{
public:
    virtual void f1(float f){
        cout << "MyBase::f1(float f)" << endl;
    }
};
class MyDerived :public MyBase{
public:
    virtual void f1(int f) override {// Compile error!
        cout << "MyDerived::f1(int f)" << endl;
    }
};
int main(){
    MyBase *pd = new MyDerived();
    pd->f1(2.4); // what will be printed?
    return 0;
}
```

final

- How can we restrict some function cannot be overridden?
  - Use `final` keyword

```cpp
#include <iostream>
using namespace std;
class MyBase{
public:
    virtual void f1(float f) {
        cout << "MyBase::f1(float f)" << endl;
    }
};
class MyDerived :public MyBase{
public:
    void f1(float f) final {
        cout << "MyDerived::f1(float f)" << endl;
    }
};
class MyDerivedDerived :public MyDerived {
public:
    void f1(float f) { // Compile error
        cout << "MyDerivedDerived::f1(float f)" << endl;
    }
};
```

final

- `final` keyword is also used for class to make the class unavailable for inheritance
  - `MyBase` class is not inheritable

```
class MyBase final{
};
class MyDerived :public MyBase {// Compile error
};
```

default

- Recall that default constructor is generated by compiler only when there is no user-define constructor
  - What if we want to use user-defined copy constructor and just default compiler-generated constructor?

```
class MyClass{
private:
    int data;
public:
    MyClass(const MyClass& obj) :data{obj.data} { // Copy constructor
    }
};

int main() {
    MyClass obj1; // Compile error: No default constructor
    return 0;
}
```

default

- We can use `default` keyword to use compiler-generated special functions like constructor, copy constructor, and assignment operator with user-defined ones

```
class MyClass
{
private:
    int data;
public:
    MyClass() = default;
    MyClass(const MyClass& obj) :data{obj.data} {

    }
};

int main(){
    MyClass obj1;
    return 0;
}
```

delete

- How can we prevent someone from copying the object of `MyClass`?
  - One way is to declare private copy constructor

```
class MyClass{
    friend MyFriend;
private:
    int data;
    MyClass(const MyClass& obj) :data{obj.data} {
    }
};

int main() {
    MyClass obj1, obj2;
    obj2 = obj1; // Compile error
    return 0;
}
```

  - However, friend class or function of `MyClass` still can copy the object of `MyClass`

delete

- You can use `delete` keyword to completely remove special functions

```
class MyClass{
    friend MyFriend;
private:
    int data;
    MyClass(const MyClass& obj)  = delete;

int main() {
    MyClass obj1, obj2;
    obj2 = obj1; // Compile error
    return 0;
}
```

delete

- How can we prevent someone from invoking `factorial()` with a parameter of other than the `int` type?

```
int factorial(int n){
    if (n <= 1)
        return n;
    else
        return n*factorial(n-1);
}

int main(){
    long f{ 0 };
    f = factorial(2);
    f = factorial(2.4);
    f = factorial('a');
    f = factorial(true);
    return 0;
}
```

delete

- You can use `delete` keyword to remove normal functions

```
int factorial(int n){
    if (n <= 1)
        return n;
    else
        return n*factorial(n-1);
}
int factorial(double) = delete;
int factorial(char) = delete;
int factorial(bool) = delete;

int main(){
    long f{ 0 };
    f = factorial(2);
    f = factorial(2.4); // Compile error
    f = factorial('a’); // Compile error
    f = factorial(true); // Compile error
    return 0;
}
```

Quiz #1

- What is the expected output? (including compile/runtime error)

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

template <class A, class B>
auto findMin(A a, B b) -> decltype(a < b ? a : b)
{
    return (a < b) ? a : b;
}

int main(){
    cout << findMin(4, 3.44) << endl;

    cout << findMin(5.4, 3) << endl;

    return 0;
}
```

Motivation on Smart Pointers

- Pointer is a powerful, but error-prune features of C++ (and C)

- Memory leak is one of the notorious problem in C++
  - Object declared in a function scope is automatically released when the function terminates
  - Memory space allocated in a function scope is **NOT** released when the function terminates
    - This is especially serious in situations where exceptions occur.

- How can we automatically release memory when it is not necessary further?

C++ Smart Pointers

- A smart pointer is an *object* that stores a pointer to a heap-allocated object
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[]`, etc.
  - These can help you manage memory
    - The smart pointer will delete the pointed-to object *at the right time* including invoking the object's destructor
      - When that is depends on what kind of smart pointer you use
    - With correct use of smart pointers, you no longer have to remember when to `delete` `new`'d memory!

A Toy Smart Pointer

- We can implement a simple one with:
  - A constructor that accepts a pointer
  - A destructor that frees the pointer
  - Overloaded `*` and `->` operators that access the pointer

ToyPtr Class Template

```
#ifndef  _TOYPTR_H_
#define  _TOYPTR_H_

template <typename T> class ToyPtr {
 public:
  ToyPtr(T *ptr) : ptr_(ptr) { }      // constructor
  ~ToyPtr() { delete ptr_; }          // destructor


  T &operator*() { return *ptr_; }   // * operator
  T *operator->() { return ptr_; }   // -> operator

 private:
  T *ptr_;                            // the pointer itself
};

#endif  // _TOYPTR_H_
```

ToyPtr Example

```cpp
#include <iostream>
#include "ToyPtr.h"

// simply struct to use
typedef struct { int x = 1, y = 2; } Point;
std::ostream &operator<<(std::ostream &out, const Point &rhs) {
  return out << "(" << rhs.x << "," << rhs.y << ")";
}

int main(int argc, char **argv) {
  // Create a dumb pointer
  Point *leak = new Point;

  // Create a "smart" pointer (OK, it's still pretty dumb)
  ToyPtr<Point> notleak(new Point);

  std::cout << "     *leak: " << *leak << std::endl;
  std::cout << "   leak->x: " << leak->x << std::endl;
  std::cout << "  *notleak: " << *notleak << std::endl;
  std::cout << "notleak->x: " << notleak->x << std::endl;

  return EXIT_SUCCESS;
}
```

What Makes This a Toy?

- Can’t handle:
  - Arrays
  - Copying
  - Reassignment
  - Comparison
  - … plus many other subtleties…
- Luckily, others have built non-toy smart pointers for us in STL!

Smart Pointers in Modern C++

- Smart Pointers
  - `std::unique_ptr`
  - Reference counting
  - `std::shared_ptr` and `std::weak_ptr`

Refresher: ToyPtr Class Template

```
#ifndef _TOYPTR_H_
#define _TOYPTR_H_

template <typename T> class ToyPtr {
 public:
  ToyPtr(T *ptr) : ptr_(ptr) { }      // constructor
  ~ToyPtr() { delete ptr_; }          // destructor

  T &operator*() { return *ptr_; }   // * operator
  T *operator->() { return ptr_; }   // -> operator

 private:
  T *ptr_;                            // the pointer itself
};

#endif  // _TOYPTR_H_
```

`std::unique_ptr`

- A `unique_ptr` ***takes ownership*** of a pointer
  - Part of C++'s standard library (C++11)
  - A template: template parameter is the type that the "owned" pointer references (*i.e.* the `T` in pointer type `T*`)
  - Its destructor invokes `delete` on the owned pointer
    - Invoked when `unique_ptr` object is `delete`'d or falls out of scope

Using `unique_ptr`

```
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::unique_ptr
#include <cstdlib>    // for EXIT_SUCCESS

void Leaky() {
  int *x = new int(5);  // heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, therefore leak

void NotLeaky() {
  std::unique_ptr<int> x(new int(5));  // wrapped, heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, but no leak

int main(int argc, char **argv) {
  Leaky();
  NotLeaky();
  return EXIT_SUCCESS;
}
```

Why are `unique_ptr`s useful?

- If you have many potential exits out of a function, it's easy to forget to call `delete` on all of them
  - Especially, an exception handling is the case
  - `unique_ptr` will `delete` its pointer when it falls out of scope
  - Thus, a `unique_ptr` also helps with *exception safety*

```
void NotLeaky() {
  std::unique_ptr<int> x(new int(5));
  ...
  // lots of code, including several returns
  // lots of code, including potential exception throws
  ...
}
```

`unique_ptr` Operations

```
#include <memory>    // for std::unique_ptr
#include <cstdlib>   // for EXIT_SUCCESS

using namespace std;
typedef struct { int a, b; } IntPair;

int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));

  int *ptr = x.get(); // Return a pointer to pointed-to object
  int val = *x;       // Return the value of pointed-to object

  // Access a field or function of a pointed-to object
  unique_ptr<IntPair> ip(new IntPair);
  ip->a = 100;

  // Deallocate current pointed-to object and store new pointer
  x.reset(new int(1));

  ptr = x.release();  // Release responsibility for freeing
  delete ptr;
  return EXIT_SUCCESS;
}
```

unique_ptrs Cannot Be Copied

- std::unique_ptr has disabled its copy constructor and assignment operator
  - You cannot copy a unique_ptr, helping maintain "uniqueness" or "ownership"

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

int main(int argc, char **argv) {
  std::unique_ptr<int> x(new int(5));

  std::unique_ptr<int> y(x);           // compile error

  std::unique_ptr<int> z;

  z = x;                               // compile error

  return EXIT_SUCCESS;
}
```

Transferring Ownership

- Use `reset()` and `release()` to transfer ownership
  - `release` returns the pointer, sets wrapped pointer to `nullptr`
  - `reset` `delete`'s the current pointer and stores a new one

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y(x.release());  // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z.reset(y.release());

  return EXIT_SUCCESS;
}
```

`unique_ptr` and STL

- `unique_ptr`s *can* be stored in STL containers
  - Wait, what? STL containers like to make lots of copies of stored objects and `unique_ptr`s cannot be copied…
- Move semantics to the rescue!
  - When supported, STL containers will *move* rather than *copy*
    - `unique_ptr`s support move semantics
- You will learn move semantics in the next lecture

unique_ptr and STL Example

```
int main(int argc, char **argv) {
  std::vector<std::unique_ptr<int> > vec;

  vec.push_back(std::unique_ptr<int>(new int(9)));
  vec.push_back(std::unique_ptr<int>(new int(5)));
  vec.push_back(std::unique_ptr<int>(new int(7)));

  int z = *vec[1];
  std::cout << "z is: " << z << std::endl;

  // compile error
  std::unique_ptr<int> copied = vec[1];

  // OK
  std::unique_ptr<int> moved = std::move(vec[1]);
  std::cout << "*moved: " << *moved << std::endl;
  std::cout << "vec[1].get(): " << vec[1].get() << std::endl;

  return EXIT_SUCCESS;
}
```

`unique_ptr` and Arrays

- `unique_ptr` can store arrays as well
  - Will call `delete[]` on destruction

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

using namespace std;

int main(int argc, char **argv) {
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```

Quiz #2

- What is the expected output? (including compile/runtime error)

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

int main()
{
    A *ap = (A *)0x1234; // Never write code like this. This is only for quiz
    unique_ptr<A> p1(ap);
    p1->show();

    // returns the memory address of p1
    cout << p1.get() << endl;

    // transfers ownership to p2
    unique_ptr<A> p2 = move(p1);
    p2->show();
    cout << p1.get() << endl; cout << p2.get() << endl;

    // transfers ownership to p3
    unique_ptr<A> p3 = move(p2);
    p3->show();
    cout << p1.get() << endl; cout << p2.get() << endl; cout << p3.get() << endl;

    p3.release();
    return 0;
}
```

Reference Counting

- Reference counting is a technique for managing resources by counting and storing the number of references (*i.e.* pointers that hold the address) to an object

std::shared_ptr

- `shared_ptr` is similar to `unique_ptr` but we allow shared objects to have multiple owners
  - The copy/assign operators are not disabled and *increment* or *decrement* reference counts as needed
    - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is 2
  - When a `shared_ptr` is destroyed, the reference count is *decremented*
    - When the reference count hits 0, we `delete` the pointed-to object!
  - You can see the ref. counter of `shared_ptr` by calling use_count() member function

shared_ptr Example

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr

int main(int argc, char **argv) {
  std::shared_ptr<int> x(new int(10));  // ref count: 1

  // temporary inner scope
  {
    std::shared_ptr<int> y = x;         // ref count: 2
    std::cout << *y << std::endl;
  }

  std::cout << *x << std::endl;         // ref count: 1

  return EXIT_SUCCESS;
}                                       // ref count: 0
```

`shared_ptr`s and STL Containers

- Even simpler than `unique_ptr`s
  - Safe to store `shared_ptr`s in containers, since copy/assign maintain a shared reference count

```
vector<std::shared_ptr<int> > vec;

vec.push_back(std::shared_ptr<int>(new int(9)));
vec.push_back(std::shared_ptr<int>(new int(5)));
vec.push_back(std::shared_ptr<int>(new int(7)));

int &z = *vec[1];
std::cout << "z is: " << z << std::endl;

std::shared_ptr<int> copied = vec[1];  // works! Incr. ref. cnt
std::cout << "*copied: " << *copied << std::endl;

// works! no change to ref. faster than copy
std::shared_ptr<int> moved = std::move(vec[1]);
std::cout << "*moved: " << *moved << std::endl;
std::cout << "vec[1].get(): " << vec[1].get() << std::endl;
```

Cycle of `shared_ptr`s

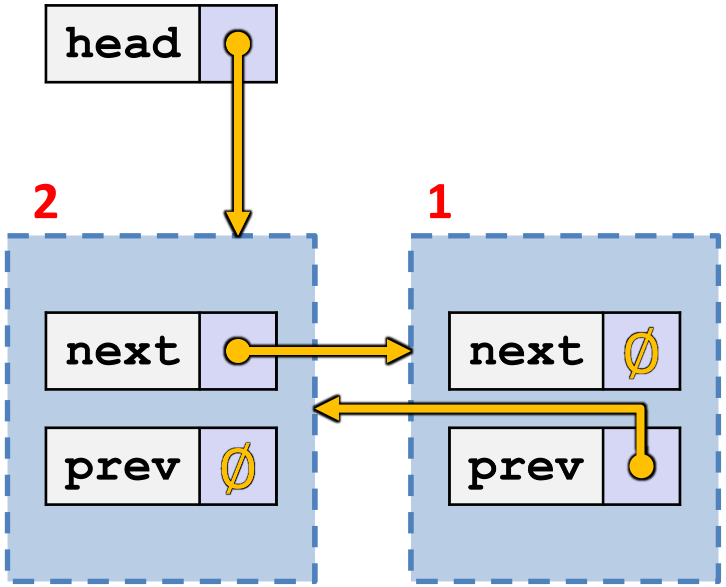
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
  shared_ptr<A> next;
  shared_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```



- What happens when we `delete head`?

`std::weak_ptr`

- `weak_ptr` is similar to a `shared_ptr` but doesn't affect the reference count
  - Can *only* "point to" an object that is managed by a `shared_ptr`
  - Not *really* a pointer – can't actually dereference unless you "get" its associated `shared_ptr`
  - Because it doesn't influence the reference count, `weak_ptr`s can become "*dangling*"
    - Object referenced may have been `delete`'d
    - But you can check to see if the object still exists

- Can be used to break our cycle problem!

Breaking the Cycle with `weak_ptr`

weakcycle.cc

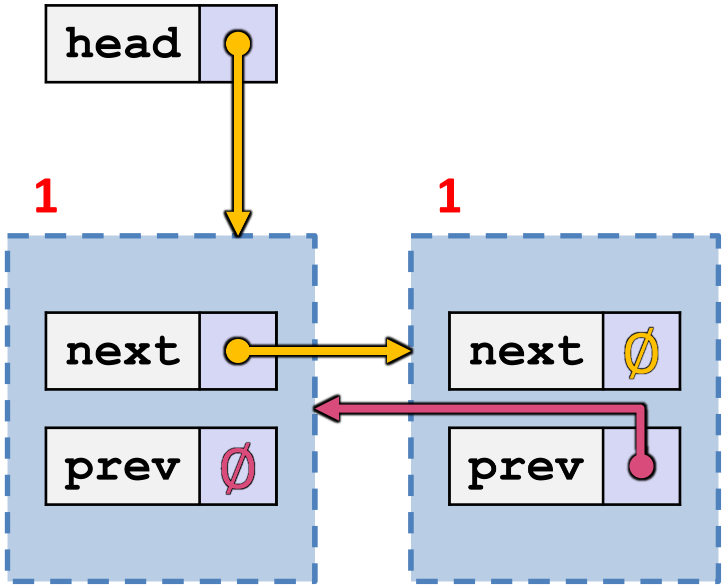
```
#include <cstdlib>
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;
};

int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;

  return EXIT_SUCCESS;
}
```



- Now what happens when we `delete head`?

Using a weak_ptr

```
#include <cstdlib>    // for EXIT_SUCCESS
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::shared_ptr, std::weak_ptr

int main(int argc, char **argv) {
  std::weak_ptr<int> w;

  {  // temporary inner scope
    std::shared_ptr<int> x;
    {  // temporary inner-inner scope
      std::shared_ptr<int> y(new int(10));
      w = y;
      x = w.lock();  // returns "promoted" shared_ptr
      std::cout << *x << std::endl;
    }
    std::cout << *x << std::endl;
  }
  std::shared_ptr<int> a = w.lock();
  std::cout << a << std::endl;

  return EXIT_SUCCESS;
}
```

Quiz #3

- What is the expected output? (including compile/runtime error)

```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    std::shared_ptr<double>  a(new double);
    *a = 3.1415;

    std::cout << "Use count: " << a.use_count() << '\n'; // A
    std::shared_ptr<double> a2(a);
    std::cout << "Use count: " << a2.use_count() << '\n'; // B

    std::weak_ptr<double> w(a2);
    std::cout << "Use count: " << w.use_count() << '\n'; // C

    w.reset();
    std::cout << "Use count: " << w.use_count() << '\n'; // D

    a2.reset();
    std::cout << "Use count: " << a2.use_count() << '\n'; // E

    std::cout << "Use count: " << a.use_count() << '\n'; // F
    return 0;
}
```

Summary of Smart Pointers

- A `unique_ptr` ***takes ownership*** of a pointer
  - Cannot be copied, but can be moved
  - `get()` returns a copy of the pointer, but is dangerous to use; better to use `release()` instead
  - `reset()` `delete`s old pointer value and stores a new one
- A `shared_ptr` allows shared objects to have multiple owners by doing *reference counting*
  - `delete`s an object once its reference count reaches zero
- A `weak_ptr` works with a shared object but doesn't affect the reference count
  - Can't actually be dereferenced, but can check if the object still exists and can get a `shared_ptr` from the `weak_ptr` if it does

New features in C++11 (and C++14)

vector<vector<int>>

=default, =delete

atomic<T>

auto f() -> int

user-defined literals

thread_local

array<T, N>

C++11

decltype

vector<LocalType>

noexcept

regex

initializer lists

extern template

constexpr

raw string literals

async

unordered_map<int, string>

template aliases

nullptr

auto i = v.begin();

delegating constructors

rvalue references (move semantics)

lambdas []{ foo(); }

override, final

variadic templates template <typename T...>

static_assert(x)

function<>

future<T>

unique_ptr<T> shared_ptr<T> weak_ptr<T>

thread, mutex

for (x : coll)

strongly-typed enums enum class E {...};

tuple<int, float, string>