Object-Oriented Programming: Modern C++

Lectured by Ming-Te Chi 紀明德

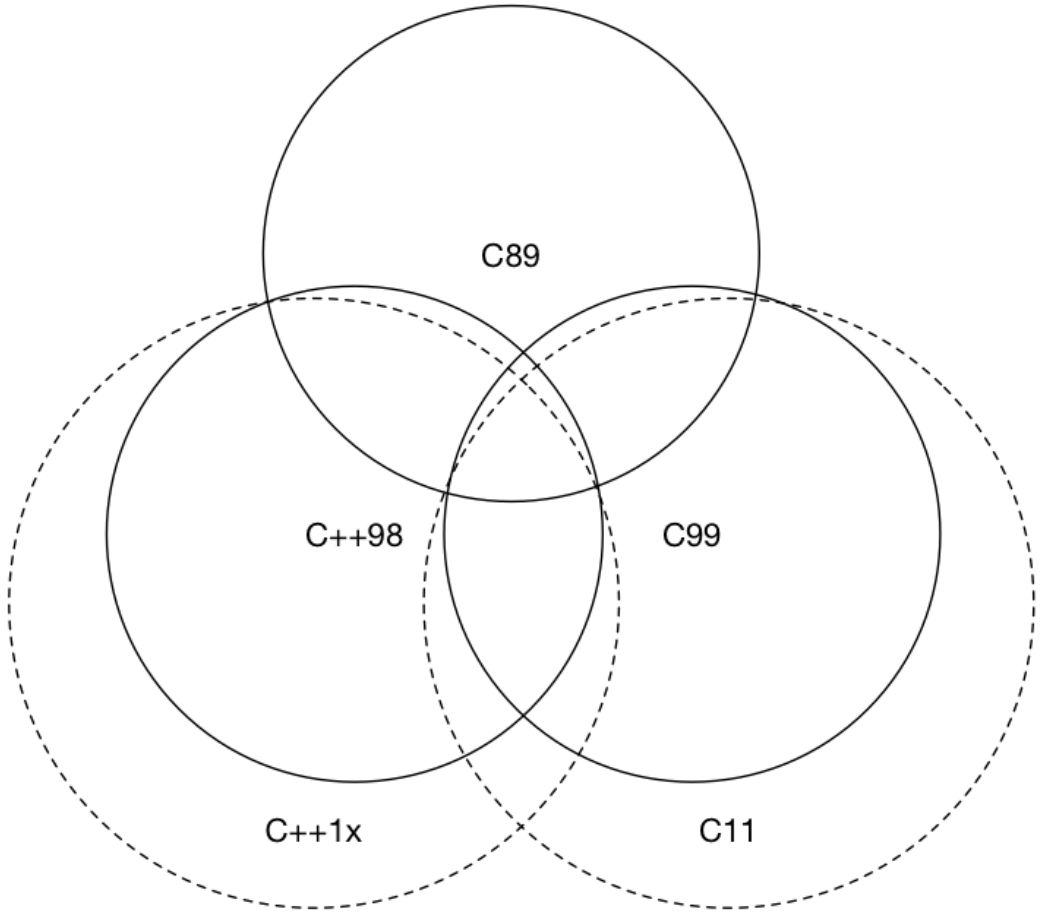
First Semester, 2022

Computer Science Department
National Chengchi University

Slides credited from 李蔡彥 and 廖峻鋒

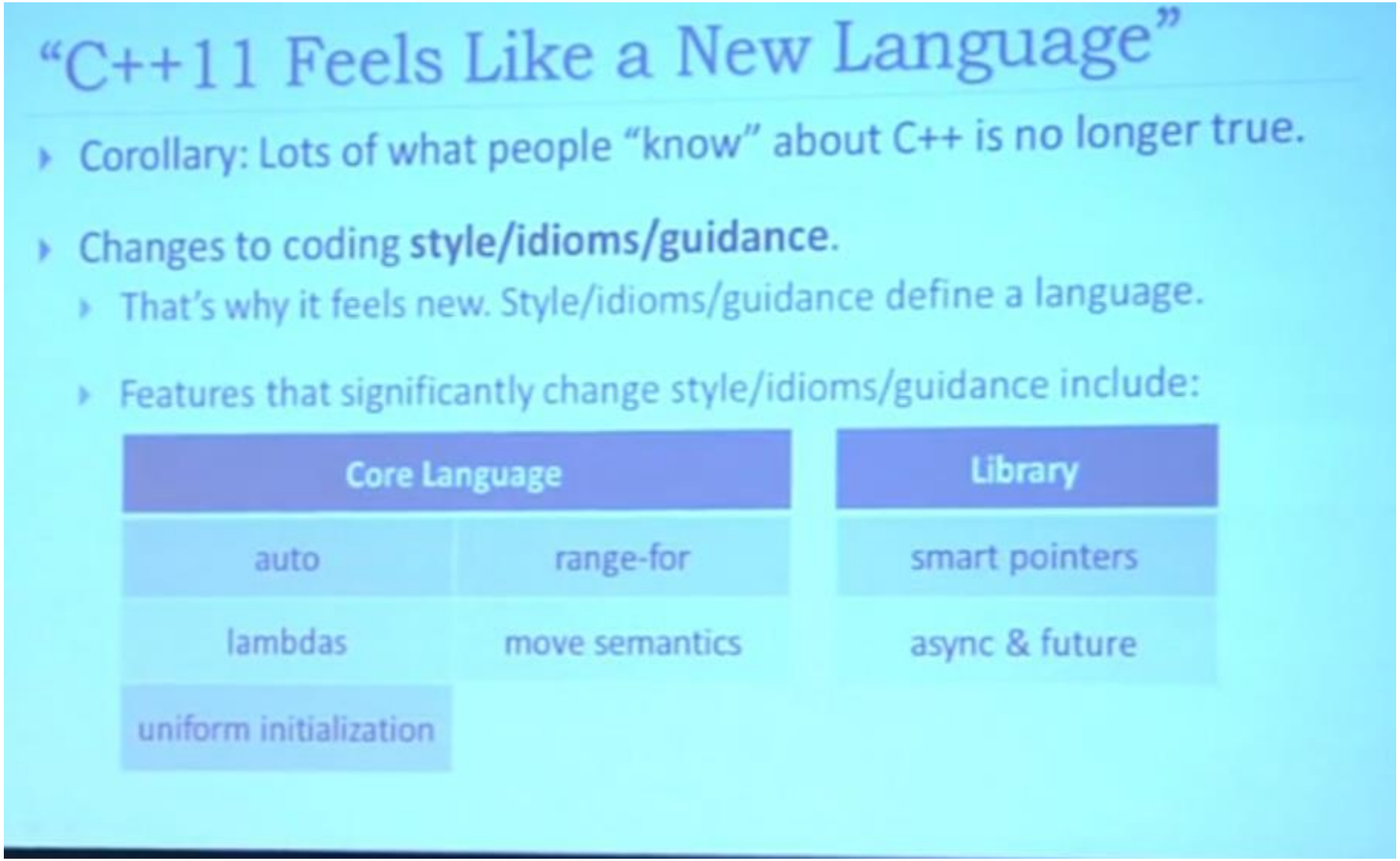
Modern C++

- Modern C++ (C++11/14/17/20)



Herb Sutter: (Not Your Father's) C++

Apr 15, 2012



nullptr

- C++11 introduces nullptr which is a special value denoting a null pointer. This should be used instead of 0 or NULL to indicate a null pointer.

```
void foo(char *);
void foo(int);

int main() {
    foo(0);          // will call foo(int)
    // foo(NULL);    // call of overloaded 'foo(NULL)' is ambiguous
    foo(nullptr);    // will call foo(char*)


}
```

constexpr

- Optimize and embed const expressions into the program at compile-time, it will increase the performance of the program.

```
constexpr int len_2_constexpr = 1 + 2 + 3;
```

```
constexpr int fibonacci(const int n) {
    return n == 1 || n == 2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}
```

```
std::cout << fibonacci(10) << std::endl;
```

uniform initialization

```
std::string s1("test");   // direct initialization
std::string s2 = "test";  // copy initialization
```

- To uniformly initialize objects regardless of their type, use the brace-initialization form {} that can be used for both direct initialization and copy initialization.

```
T object {other};   // direct list initialization
T object = {other}; // copy list initialization

int i { 42 };
double d { 1.2 };
int arr1[3] { 1, 2, 3 };
int* arr2 = new int[3]{ 1, 2, 3 };
std::vector<int> v { 1, 2, 3 };
std::map<int, std::string> m { {1, "one"}, { 2, "two" }};
```

auto

- For variables, specifies that the type of the variable that is being declared will be automatically deduced from its initializer.

```
auto i = 5;                    // i as int
auto arr = new auto(10); // arr as int *
```

```
//c++20 auto can even be used as function arguments.
int add(auto x, auto y) {
    return x+y;
}

auto i = 5; // type int
auto j = 6; // type int
std::cout << add(i, j) << std::endl;
```

auto

Before C++11

```
std::vector<std::string> user_name = {"sam", "li", "chi"};
std::vector<std::string>::const_iterator it =
        std::find(user_name.begin(), user_name.end(), "sam");

if (it != user_name.end()) {
    // Do something...
}
```

After C++11

```
std::vector<std::string> user_name = {"sam", "li", "chi"};
auto it = std::find(user_name.begin(), user_name.end(), "sam");

if (it != user_name.end()) {
    // Do something...
}
```

decltype

- Inspects the declared type of an entity or the type and value category of an expression.

```
decltype(1.0 + 1) the_double = 0.0;
```

```
//c++14
template <typename Container, typename Index>
decltype(auto)
DoSomething(Container& c, Index i)
{
    return c[i];
}
```

Range-Based for Loops (since C++11)

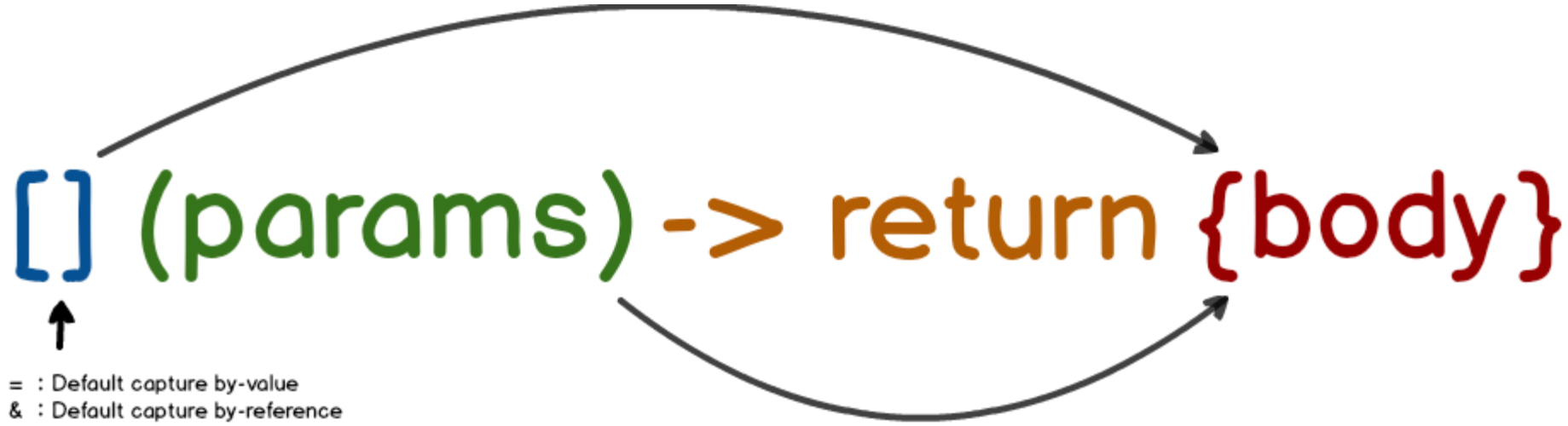
- Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

```
std::vector<int> vi = {1, 2, 3, 4, 5};
//before c++11
for (int i = 0; i < vi.size(); ++i) {
    printf("i = %d\n", vi[i]);
}

//since c++11
for (auto i : vi) {
    printf("i = %d\n", i); // read-only
}
for (auto &element : vi) {
        element += 1; // writeable
}
```

Lambda expressions (since C++11)

- Anonymous Function Object



Function Object / sort descending

[code]

```
struct myclass {
  bool operator() (int a, int b) { return a > b; } // 降序排列
} myobject;

int main() {
  std::vector<int> v = {5, 4, 1, 7, 3, 8, 9, 10, 6, 2};
  std::sort(v.begin(), v.end(), myobject);

  return 0;
}
```

Anonymous Function Object / sort descending

```
int main() {
  std::vector<int> v = {5, 4, 1, 7, 3, 8, 9, 10, 6, 2};
  std::sort(v.begin(), v.end(), [] (int a, int b) { return a > b; } );

  return 0;
}
```

```
template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );


The signature of the comparison function should be equivalent to the following:
 bool comp(const Type1 &a, const Type2 &b);
```

Lambda: Capture-default by-value

```
std::vector<int> sources = {1, 2, 3, 4, 5, 6, 7, 8};
int minimal = 6;
auto count = std::count_if(begin(sources), end(sources),
                           [=](int s)
                           {
                               return s > minimal;
                           });
```

individual capture

```
std::vector<int> sources = {1, 2, 3, 4, 5, 6, 7, 8};
int minimal = 6;
auto count = std::count_if(begin(sources), end(sources),
                           [minimal](int s)
                           {
                               return s > minimal;
                           });
```

Lambda: Capture by-reference

```
std::vector<int> iv = {1, 1, 1, 2, 3, 4, 5, 6, 8};
int call_count = 0;

auto it = std::find_if(begin(iv), end(iv),
                       [&](int i)
{
    ++call_count;
    return i > 6;
}
```

Move Semantics and Rvalue References

```
#include <vector>
struct A {
    int a;
    int x;
};
int main() {
    using namespace std;
    A a1;
    A a2;
    vector<A> va;
    va.push_back(a1);                  //vector<T>::push_back(T&)
    va.push_back(std::move(a2));   //vector<T>::push_back(T&&)
}
```

Thread

[code]

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

void f1() { cout<<"Hello ";}

struct F2 {
    void operator()() { cout<<"Parallel World\n"; }
};

int main()
{
    thread t1 { f1 };   //f1在執行緒t1執行
    thread t2 { F2() }; //F2在執行緒t2執行

    t1.join();   //等待執行緒t1完成。
    t2.join();   //等待執行緒t2完成。

    return 0;
}
```

Thread: race problem

[code]

```
void f1() { cout<<"Hello ";}
struct F2 {
    string &s;
    F2(string ins) :s(ins) {};
    void operator()() { cout<<s; }
};
int main() {
    string s2{"Parallel "};
    string s3{"World\n"};

    thread t1 { f1 };
    thread t2 { F2{s2} };
    thread t3 { [&s3]{cout<<s3;} };

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

Result:
Parallel Hello World
Or
Hello World
Parallel

A Taste of C++ 11 (Herb Sutter: (Not Your Father's) C++)

[code]

```
string flip(string s) {
    reverse(begin(s), end(s));
    return s;
}

int main()
{
    vector<future<string>> v;

    v.push_back( async([]{ return flip(" ,olleH");}) );
    v.push_back( async([]{ return flip(" nredoM");}) );
    v.push_back( async([]{ return flip("\n!++C");}) );

    for(auto& e: v) {
        cout<< e.get();
    }



    return 0;
}
```