Object-Oriented Programming and Data Structures

COMP2012: Pointer, Reference, New C++11 Features, C++ Class Revision & const-ness

Cecia Chan
Brian Mak
Dimitris Papadopoulos

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

Why Take This Course?

You have taken COMP1021/1022P and COMP2011. So you can program already, right?

- Think about this: You have been learning English for many years, but can you write a novel?
- You basically have learned the C part of C++ in COMP2011 with a brief introduction to C++ classes, and you can write small C++ programs.
- But what if you are going to write a large program, probably with a team of programmers?

In this course, you will learn the essence of OOP with some new C++ constructs with an aim to write large softwares.

Part I

Quick Review: Reference and Pointer

Variable, Reference Variable, Pointer Variable

```
#include <iostream>          /* File: confusion.cpp */
using namespace std;

int x = 5;                   // An int variable
int& xref = x;               // A reference variable: xref is an alias of x
int* xptr = &x;              // A pointer variable: xptr points to x

void xprint()
{
    cout << hex << endl; // Print numbers in hexadecimal format
    cout << "x =      " << x     << "\t\tx address     = " << &x << endl;
    cout << "xref =   " << xref  << "\t\txref address  = " << &xref << endl;
    cout << "xptr =   " << xptr  << "\txptr address  = "   << &xptr << endl;
    cout << "*xptr = " << *xptr << endl;
}

int main()
{
    x += 1; xprint();
    xref += 1; xprint();
    xptr = &xref; xprint(); // Now xptr points to xref

    return 0;
}
```

Pointer vs. Reference

Reference can be thought as a special kind of pointer, but there are 3 big differences:

1. A pointer can point to nothing (`nullptr`), but a reference is always bound to an object.
2. A pointer can point to different objects at different times (through assignments). A reference is always bound to the same object.

   Assignments to a reference does not change the object it refers to but only the value of the referenced object.
3. The name of a pointer refers to the pointer itself. The `*` or `->` operators have to be used to access the underlying object it points to.

   The name of a reference always refers to the object. There are no special operators for references.

Part II

Some New Features in C++11

A List of New Features in C++11

- uniform and general initialization using { }-list ⋆
- type deduction of variables from initializer: auto
  — NOT ALLOWED TO USE IN COMP2011/2012
- prevention of narrowing ⋆
- generalized and guaranteed constant expressions: constexpr
- Range-for-statement ⋆
- null pointer keyword: nullptr ⋆
- scoped and strongly typed enums: enum_class
- rvalue references, enabling move semantics †
- lambdas or lambda expressions ⋆
- support for unicode characters
- long long integer type
- delegating constructors †
- in-class member initializers †
- explicit conversion operators †
- override control keywords: override and final †

General Initialization Using { }-Lists

- In the past, you always initialize variables using the assignment operator =.

Example: = Initializer

```
int x = 5;
float y = 9.8;
int& xref = x;
int a[] = {1, 2, 3};
```

- C++11 allows the more uniform and general curly-brace-delimited initializer list.

Example: { } Initializer

```
int x = {5};        // = here is optional
float y {9.8};
int& xref {x};
int a[] {1, 2, 3};
```

Initializer Example 1

```cpp
#include <iostream>      /* File: initializer1.cpp */
using namespace std;

int main()
{
    int w = 3.4;
    int x1 {6};
    int x2 = {8};        // = here is optional
    int y {'k'};
    int z {6.4};         // Error!

    cout << "w = " << w << endl;
    cout << "x1 = " << x1 << endl << "x2 = " << x2 << endl;
    cout << "y = " << y << endl << "z = " << z << endl;

    int& ww = w;
    int& www {ww}; www = 123;
    cout << "www = " << www << endl;
    return 0;
}
```

```
initializer1.cpp:10:15: error: narrowing conversion of 6.4000000000000004e+0
from double to int inside { } [-Wnarrowing]
     int z {6.4};
```

Initializer Example 2

```
#include <iostream>     /* File: initializer2.cpp */
using namespace std;

int main()
{
    const char s1[] = "Steve Jobs";
    const char s2[] {"Bill Gates"};
    const char s3[] = {'h', 'k', 'u', 's', 't', '\0'};
    const char s4[] {'h', 'k', 'u', 's', 't', '\0'};

    cout << "s1 = " << s1 << endl;
    cout << "s2 = " << s2 << endl;
    cout << "s3 = " << s3 << endl;
    cout << "s4 = " << s4 << endl;
    return 0;
}
```

Differences Between the = and { } Initializers

- The { } initializer is more restrictive: it doesn't allow conversions that lose information — narrowing conversions.
- The { } initializer is more general as it also works for:
  - arrays
  - other aggregate structures
  - class objects (we'll talk about that later)

Range-for Statements

- In the past, you write a for-loop by
  - initializing an index variable,
  - giving an ending condition, and
  - writing some post-processing that involves the index variable.

Example: Traditional for-Loop

```
for (int k = 0; k < 5; ++k)
    cout << k*k << endl;
```

- C++11 adds a more flexible range-for syntax that allows looping through a sequence of values specified by a list.

Example: Range-for-Loops

```
for (int k : { 0, 1, 2, 3, 4 })
    cout << k*k << endl;

for (int k : { 1, 19, 54 }) // Numbers need not be successive
    cout << k*k << endl;
```

Range-for Example

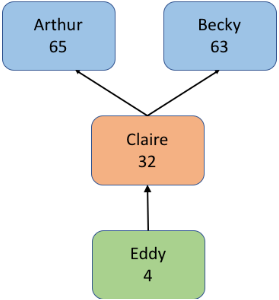
```
#include <iostream>      /* File : range-for.cpp */
using namespace std;

int main()
{
    cout << "Square some numbers in a list" << endl;
    for (int k : {0, 1, 2, 3, 4})
        cout << k*k << endl;

    int range[] { 2, 5, 27, 40 };

    cout << "Square the numbers in range" << endl;
    for (int k : range)  // Won't change the numbers in range
        cout << k*k << endl;

    cout << "Print the numbers in range" << endl;
    for (int v : range) cout << v << endl;

    for (int& x : range) // Double the numbers in range in situ
        x *= 2;

    cout << "Again print the numbers in range" << endl;
    for (int v : range) cout << v << endl;
    return 0;
}
```

Part III

A Revision Example: Person and Family

A Revision Example: Person & Family

- It consists of the class Person, from which families are built.
- A person, in general, has at most 1 child, and his/her father and mother may or may not be known.
- The information of his/her family includes him/her and his/her parents and grandparents from both of his/her parents.



Revision Example: Expected Output

```
Name: Arthur
Father: unknown
Mother: unknown
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Becky
Father: unknown
Mother: unknown
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Claire
Father: Arthur
Mother: Becky
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Eddy
Father: unknown
Mother: Claire
Grand Fathers: unknown, Arthur
Grand Mothers: unknown, Becky
```

Revision Example: Person Class — Header File

```
#include <iostream>     /* File: person.h */
using namespace std;

class Person
{
  private:
    char* _name;
    int _age;
    Person *_father, *_mother, *_child;

  public:
    Person(const char* my_name, int my_age, Person* my_father = nullptr,
           Person* my_mother = nullptr, Person* my_child = nullptr);
    ~Person();

    Person* father() const;
    Person* mother() const;
    Person* child() const;
    void print_age() const;
    void print_name() const;
    void print_family() const;

    void have_child(Person* baby) ;
};
```

```cpp
#include "person.h"     /* File: person.cpp */
#include <cstring>

Person::Person(const char* my_name, int my_age, Person* my_father,
               Person* my_mother, Person* my_child)
{
    _name = new char [strlen(my_name)+1];
    strcpy(_name, my_name);
    _age = my_age;
    _father = my_father;
    _mother = my_mother;
    _child = my_child;
};

Person::~Person() { delete [] _name; }

Person* Person::father() const { return _father; }

Person* Person::mother() const { return _mother; }

Person* Person::child() const { return _child; }

void Person::have_child(Person* baby) { _child = baby; }
```

```
void Person::print_age() const { cout << _age; }

void Person::print_name() const
{
    cout << (_name ? _name : "unknown");
}




// Helper function
void print_parent(Person* parent)
{
    if (parent)
        parent->print_name();
    else
        cout << "unknown";
}
```

```
void Person::print_family() const
{
    Person *f_grandfather = nullptr, *f_grandmother = nullptr,
        *m_grandfather = nullptr, *m_grandmother = nullptr;

    if (_father) {
        f_grandmother = _father->mother();
        f_grandfather = _father->father();
    }

    if (_mother) {
        m_grandmother = _mother->mother();
        m_grandfather = _mother->father();
    }

    cout << "Name: "; print_name(); cout << endl;
    cout << "Father: "; print_parent(_father); cout << endl;
    cout << "Mother: "; print_parent(_mother); cout << endl;

    cout << "Grand Fathers: "; print_parent(f_grandfather);
    cout << ", "; print_parent(m_grandfather); cout << endl;
    cout << "Grand Mothers: "; print_parent(f_grandmother);
    cout << ", "; print_parent(m_grandmother); cout << endl;
}
```

Revision Example: Family Building Test Program

```
#include "person.h"     /* File: family.cpp */

int main()
{
    Person arthur("Arthur", 65, nullptr, nullptr, nullptr);
    Person becky("Becky", 63, nullptr, nullptr, nullptr);
    Person claire("Claire", 32, &arthur, &becky, nullptr);
    Person eddy("Eddy", 4, nullptr, &claire, nullptr);

    arthur.have_child(&claire);
    becky.have_child(&claire);
    claire.have_child(&eddy);

    arthur.print_family(); cout << endl;
    becky.print_family();  cout << endl;
    claire.print_family(); cout << endl;
    eddy.print_family();   cout << endl;
    return 0;
}
```

Part IV

General Remarks on C++ Classes

Structure vs. Class

In C++ , structures are special classes and they may have member functions. By default,

$$\text{struct } \{ \ldots \} \equiv \text{class } \{ \text{ public: } \ldots \}$$
$$\text{class } \{ \ldots \} \equiv \text{struct } \{ \text{ private: } \ldots \}$$

```
#include <iostream>     /* File: struct/person.h */
using namespace std;
struct Person
{
    char* _name;
    int _age;
    Person *_father, *_mother, *_child;
    Person(const char* my_name, int my_age, Person* my_father = nullptr,
           Person* my_mother = nullptr, Person* my_child = nullptr);
    ~Person();
    Person* father() const;
    Person* mother() const;
    Person* child() const;
    void print_age() const;
    void print_name() const;
    void print_family() const;
    void have_child(Person* baby) ;
};
```

Class Name: Name Equivalence

- A class definition introduces a new abstract data type.
- C++ relies on name equivalence (and not structure equivalence) for class types.

```
class X { int a; };
class Y { int a; };
class W { int a; };
class W { int b; };  // Error, double definition

X x;
Y y;

x = y;  // Error: type mismatch
```

Class Data Members

Data members can be any basic type, or any user-defined types if they are already declared.

Below are special cases:

- A class name can be used inside its own definition for a pointer to an object of the class:

```
class Cell
{
    int info;
    Cell* next;
};
```

- A forward declaration of a class X can be used in the definition of another class Y to define a pointer to X:

```
class Cell;         // Forward declaration of Cell

class List
{
    int size;
    Cell* data;     // Points to a (forward-declared) Cell object
    Cell x;         // Error: Cell not defined yet!
};

class Cell          // Definition of Cell
{
    int info;
    Cell* next;
};
```

Default Initializer for Non-static Members (C++11)

```
class Complex
{
  private:
    float real = 1.3;    // Note: not allowed before C++11
    float imag {0.5};    // Use either = or { } initializer
  public:
    ...
};
```

- You are advised to initialize non-static data member values by
  - class constructors
  - class member initializer list in a constructor
  - class member functions
- Non-static data members that are not initialized by the 3 ways above will have the values of their default member initializers if they exist, otherwise their values are undefined.
- We'll talk about static vs. non-static members later. All data members you'll see most of the time are non-static.

Class Member Functions

- These are the functions declared inside the body of a class.
- They can be defined in 3 ways:

1. as inline functions within the class body. The keyword inline is optional in this case.

```
class Person
{   ...
    Person* child() const { return _child; }
    void have_child(Person* baby) { _child = baby; }
};
```

Or,

```
class Person
{   ...
    inline Person* child() const { return _child; }
    inline void have_child(Person* baby) { _child = baby; }
};
```

2. as inline functions, but outside the class body, in the same header file. In this case, the keyword inline is mandatory. It also requires the additional prefix consisting of the class name and the class scope operator ::
   ⇒ to enhance readability especially when the class body consists of a few lines of code.

```
/* File: person.h */
class Person
{   ...
    inline Person* child() const;
    inline void have_child(Person* baby);
};

inline Person* Person::child() const { return _child; }
inline void Person::have_child(Person* baby) { _child = baby; }
```

3. as non-inline functions, outside the class body, in a separate implementation .cpp file. Then add the prefix consisting of the class name and the class scope operator ::
   ⇐ any benefits of doing this?

```
/* File: person.h */
class Person
{   ...
    Person* child() const;
    void have_child(Person* baby);
};

/* File: person.cpp */
Person* Person::child() const { return _child; }
void Person::have_child(Person* baby) { _child = baby; }
```

Class Scope and Scope Operator ::

- C++ uses lexical (static) scope rules: the binding of name occurrences to declarations are done statically at compile-time.
- Identifiers declared inside a class definition are under its scope.
- To define the members functions outside the class definition, prefix the identifier with the class scope operator ::
- e.g., `temperature::kelvin()`, `temperature::celsius()`

```
int height = 10;
class Weird
{
    short height;
    Weird() { height = 5; }
};
```

Q1 : Which "height" is used in `Weird::Weird()`?

Q2 : Can we access the global height inside the Weird class body?

This Pointer

- Each class member function implicitly contains a pointer of its class type named "this".
- When an object calls the function, this pointer is set to point to the object.
- For example, after compilation, the member function `Person::have_child(Person* baby)` of Person will be translated to a unique global function by adding a new argument:

```
void Person::have_child(Person* this, Person* baby)
{
    this->_child = baby;
}
```

- The call, `becky.have_child(&eddy)` becomes

  `Person::have_child(&becky, &eddy)`.

Example: Return an Object by this — complex.h

```
class Complex              /* File: complex.h */
{
  private:
    float real; float imag;

  public:
    Complex(float r, float i) { real = r; imag = i; }
    void print() const { cout << "(" << real << " , " << imag << ")" << endl; }

    Complex add1(const Complex& x)  // Return by value
    {
        real += x.real; imag += x.imag;
        return (*this);
    }
    Complex* add2(const Complex& x) // Return by value using pointer
    {
        real += x.real; imag += x.imag;
        return this;
    }
    Complex& add3(const Complex& x) // Return by reference
    {
        real += x.real; imag += x.imag;
        return (*this);
    }
};
```

Example: Return an Object by this — complex-test.cpp

```
#include <iostream>      /* File: complex-test.cpp */
using namespace std;
#include "complex.h"

void f(const Complex a) { a.print(); }   // const Complex a  = u
void g(const Complex* a) { a->print(); } // const Complex* a = &u
void h(const Complex& a) { a.print(); }  // const Complex& a = u

int main()
{
    // Check the parameter passing methods
    Complex u(4, 5); f(u); g(&u); h(u);

    // Check the parameter returning methods
    Complex w(10, 10); cout << endl << endl;
    Complex x(4, 5); (x.add1(w)).print();    // Complex  temp = *this = x
    Complex y(4, 5); (y.add2(w))->print();   // Complex* temp =  this = &y
    Complex z(4, 5); (z.add3(w)).print();    // Complex& temp = *this = z

    cout << endl << endl;          // What is the output now?
    Complex a(4, 5); a.add1(w).add1(w).print();   a.print(); cout << endl;
    Complex b(4, 5); b.add2(w)->add2(w)->print(); b.print(); cout << endl;
    Complex c(4, 5); c.add3(w).add3(w).print();   c.print();
    return 0;
}
```

Return-by-Value and Return-by-Reference

There are 2 ways to pass parameters to a function

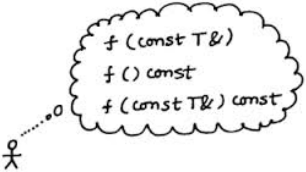
- pass-by-value (PBV)
- pass-by-reference (PBR)
  - lvalue reference: that is what you learned in the past and we'll keep just saying *reference* for lvalue reference.
  - rvalue reference (C++11)

Similarly, you may return from a function by returning an object's

- value: the function will make a separate copy of the object and return it. Changes made to the copy have no effect on the original object.
- (lvalue) reference: the object itself is passed back! Any further operations on the returned object will directly modify the original object as it is the same as the returned object.
- rvalue reference: we'll talk about this later.

Part V

const-ness



const

- const, in its simplest usage, is used to express a user-defined constant — a value that can't be changed.

  ```
  const float PI = 3.1416;
  ```

- Some people like to write const identifiers in capital letters.
- In the old days, constants are defined by the #define preprocessor directive:

  ```
  #define PI 3.1416
  ```

  Question: Any shortcomings?

- const actually may be used to represent more than just numerical constants, but also const objects, pointers, and even member functions!
- The const keyword can be regarded as a safety net for programmers: If an object should not change, make it const.

Example: Constant Object of User-defined Types

```
/* File: const-object-date.h */

class Date       // There are problems with this code; what are they?
{
  private:
    int year, month, day;

  public:
    Date() { cin >> year >> month >> day; }
    Date(int y, int m, int d) { year = y; month = m; day = d; }

    void add_month() { month += 1; }; // Will be an inline function

    int difference(const Date& d)
    { /* Incomplete: write this function */ }

    void print()
        { cout << year << "/" << month << "/" << day << endl; }
};
```

```
#include <iostream>      /* File: const-object-date.cpp */
using namespace std;
#include "const-object-date.h"

int main()   // There are problems with this code; what are they?
{
    const Date WW2(1945, 9, 2); // World War II ending date
    Date today;
    WW2.print();
    today.print();

    // How long has it been since World War II?
    cout << "Today is " << today.difference(WW2)
         << " days after WW2" << endl;

    // What about next month?
    WW2.add_month();     // Error; do you mean today.add_month()??
    cout << today.difference(WW2) << " days by next month.\n";

    return 0;
}
```

const Member Functions

- To indicate that a class member function does not modify the class object — its data member(s), one can (and should!) place the const keyword after the argument list.

```
class Date                  /* File: const-object-date2.h */
{
  private:
    int year, month, day;

  public:
    Date() { cin >> year >> month >> day; }
    Date(int y, int m, int d) { year = y; month = m; day = d; }

    void add_month() { month += 1; }; // Will be an inline function

    int difference(const Date& d) const { /* Incomplete */ }
    void print() const
        { cout << year << "/" << month << "/" << day << endl; }
};
```

const Member Functions and this Pointer

- A const object can only call const member functions of its class.
- But a non-const object can call both const and non-const member functions of its class.
- The this pointer in const member functions points to const objects. For example,
  - ▷ `int Date::difference(const Date& d) const;` is compiled to

    ```
    int Date::difference(const Date* this, const Date& d);
    ```

  - ▷ `void Date::print() const;` is compiled to

    ```
    void Date::print(const Date* this);
    ```

- Thus, the object calling const member function becomes const inside the function and cannot be modified.

const and const Pointers

- When a pointer is used, two objects are involved:
  - the pointer itself
  - the object being pointed to
- The syntax for pointers to constant objects and constant pointers can be confusing. The rule is that
  - any const to the left of the ∗ in a declaration refers to the object being pointed to.
  - any const to the right of the ∗ refers to the pointer itself.
- It can be helpful to read these declarations from right to left.

```
/* File: const-char-ptrs1.cpp */
char c = 'Y';
char *const cpc = &c;
char const* pcc;
const char* pcc2;
const char *const cpcc = &c;
char const *const cpcc2 = &c;
```

Example: const and const Pointers

```
#include <iostream>       /* File: const-char-ptrs2.cpp */
using namespace std;

int main()
{
    char s[] = "COMP2012"; // Usual initialization in the past
    char p[] {"MATH1013"}; // C++11 style of uniform initialization

    const char* pcc {s};   // Pointer to constant char
    pcc[5] = '5';          // Error!
    pcc = p;               // OK, but what does that mean?

    char *const cpc = s;   // Constant pointer
    cpc[5] = '5';          // OK
    cpc = p;               // Error!

    const char *const cpcc = s; // const pointer to const char
    cpcc[5] = '5';         // Error!
    cpcc = p;              // Error!
    return 0;
}
```

Having a pointer-to-const pointing to a non-const object doesn't make that object a constant!

```
/* File: const-int-ptr.cpp */
int i = 151;
i += 20;     // OK

int* pi = &i;
*pi += 20;   // OK

const int* pic = &i;
*pic += 20; // Error! Can't change i through pic

pic = pi;    // OK
*pic += 20; // Error! Can't change *pi thru pic

pi = pic;    // Error: Invalid conversion from 'const int*' to 'int*'
```

const References as Function Arguments

- There are 2 good reasons to pass an argument as a reference. What are they?
- You can (and should!) express your intention to leave a reference argument of your function unchanged by making it const.
- There are 2 advantages:

1. If you accidentally try to modify the argument in your function, the compiler will catch the error.

```
void cbr(int& x) { x += 10; }          // Fine

void cbcr(const int& x) { x += 10; } // Error!
```

2. You may pass both const and non-const arguments to a function that requires a const reference parameter.

   Conversely, you may pass only non-const arguments to a function that requires a non-const reference parameter.

```
#include <iostream>
using namespace std;
void cbr(int& a) { cout << a << endl; }
void cbcr(const int& a) { cout << a << endl; }
int main()
{
    int x {50}; const int y {100};
    // Which of the following give(s) compilation error?
    cbr(x);
    cbcr(x);
    cbr(y);
    cbcr(y);
    cbr(1234);
    cbcr(1234);
}
```

Summary: Good Practice

- Objects you don't intend to change ⇒ const objects

```
const double PI = 3.1415927;
const Date handover(1, 7, 1997);
```

- Function arguments you don't intend to change
  ⇒ const arguments

```
void print_height(const Large_Obj& LO){ cout << LO.height(): }
```

- Class member functions that don't change the data members
  ⇒ const member functions

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

Summary

- Regarding which objects can call const or non-const member functions:

Calling Object	const Member Function	non-const Member Function
const Object	$\surd$	X
non-const Object	$\surd$	$\surd$

- Regarding which objects can be passed to functions with const or non-const reference/pointer arguments:

Passing Object	const Function Argument	non-const Function Argument
literal constant	$\surd$	X
const Object	$\surd$	X
non-const Object	$\surd$	$\surd$

Part VI

Local Anonymous Functions — Lambdas

Lambda Expressions (Lambdas)

Syntax: Lambda

[<capture-list>] (<parameter-list>) mutable → <return-type> { <body> }

- They are anonymous functions — functions *without* a name.
- They are usually defined locally inside functions, though global lambdas are also possible.
- The capture list (of variables) allows lambdas to use local variables that are already defined in the enclosing function.
  - [=]: capture all local variables by value.
  - [&]: capture all local variables by reference.
  - [variables]: specify only the variables to capture
  - global variables can always be used in lambdas without being captured. It is an error to capture them in lambdas.
- The return type
  - is void by default if there is no return statement.
  - is automatically inferred if there is a return statement.
  - may be explicitly specified by the → syntax.

Example: Simple Lambdas with No Captures

```
#include <iostream>      /* File : simple-lambdas.cpp */
using namespace std;

int main()
{
    // A lambda for computing squares
    int range[] =  { 2, 5, 7, 10 };
    for (int v : range)
        cout << [](int k) { return k * k; } (v) << endl;

    // A lambda for doubling numbers
    for (int& v : range) [](int& k) { return k *= 2; } (v);
    for (int v : range) cout << v << "\t";
    cout << endl;

    // A lambda for computing max between 2 numbers
    int x[3][2] = { {3, 6}, {9, 5}, {7, 1} };
    for (int k = 0; k < sizeof(x)/sizeof(x[0]); ++k)
        cout << [](int a, int b) { return (a > b) ? a : b; } (x[k][0], x[k][1])
             << endl;

    return 0;
}
```

```
1  #include <iostream>     /* File : lambda-capture.cpp */
2  using namespace std;
3  int main()
4  {
5      int sum = 0, a = 1, b = 2, c = 3;
6
7      for (int k = 0; k < 4; ++k) // Evaluate a quadratic polynomial
8          cout << [=](int x) { return a*x*x + b*x + c; } (k) << endl;
9      cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;
10
11     for (int k = 0; k < 4; ++k) // a and b are used as accumulators
12         cout << [&](int x) { a += x*x; return b += x; } (k) << endl;
13     cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;
14
15     for (int v : { 2, 5, 7, 10 }) // Only variable sum is captured
16         cout << [&sum](int x) { return sum += a*x; } (v) << endl; // Error!
17     cout << "sum = " << sum << endl;
18
19     return 0;
20 }
```

```
lambda-capture.cpp:16:47: error: variable 'a' cannot be implicitly captured
      in a lambda with no capture-default specified
        cout << [&sum](int x) { return sum += a*x; } (v) << endl;
                                              ^
```

Example: When Are Values Captured?

```
#include <iostream>      /* File : lambda-value-binding.cpp */
using namespace std;

int main()
{
    int a = 1, b = 2, c = 3;
    auto f = [=](int x) { return a*x*x + b*x + c; };

    for (int k = 0; k < 4; ++k)
        cout << f(k) << endl;
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    a = 11, b = 12, c = 13;
    for (int k = 0; k < 4; ++k)
        cout << f(k) << endl; // Will f use the new a, b, c?
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    return 0;
}
```

- The keyword auto allows one to declare a variable without a type which will be inferred automatically by the compiler.
- WARNING: You are not allowed to use auto in this course!

Example: When Are References Captured?

```
#include <iostream>      /* File : lambda-ref-binding.cpp */
using namespace std;

int main()
{
    int a = 1, b = 2, c = 3;
    auto f = [&](int x) { a *= x; b += x; c = a + b; };

    for (int k = 1; k < 3; f(k++))
        ;
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    a = 11, b = 12, c = 13;
    for (int k = 1; k < 3; f(k++)) // Will f use the new a, b, c?
        ;
    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;

    return 0;
}
```

Question: What is the printout now?

Capture by Value or Reference

- When a lambda expression captures variables by value, the values are captured by copying only once at the time the lambda is defined.
- Capture-by-value is similar to pass-by-value.
- Unlike PBV, variables captured by value cannot be modified inside the lambda unless you make it mutable.

Examples

```
/* File: mutable-lambda.cpp*/
int a = 1, b = 2;

cout << [a](int x) { return a += x; } (20) << endl; // Error!
cout << [b](int x) mutable { return b *= x; } (20) << endl; // OK!
cout << "a = " << a << "\tb = " << b << endl;
```

- Similarly, capture-by-reference is similar to pass-by-reference.

Example: Mutable Lambda with Return

```
#include <iostream>      /* File : mutable-lambda-with-return.cpp */
using namespace std;

int main()
{
    float a = 1.6, b = 2.7, c = 3.8;

    // [&, a] means all except a are captured by reference; a by value
    auto f = [&, a](int x) mutable ->int { a *= x; b += x; return c = a+b; };

    for (int k = 1; k < 3; ++k)
        cout << "a = " << a << "\tb = " << b << "\tc = " << c
             << "\tf(" << k << ") = " << f(k) << endl;

    cout << "a = " << a << "\tb = " << b << "\tc = " << c << endl;
    return 0;
}
```

- One may mix the capture-default [=] or [&] with explicit variable captures as in [&, a] above.
- In this case, all variables but a are captured by reference while a is captured by value.
- But the exceptions must be given after [=] or [&].