COMP 2012H Honors Object-Oriented Programming and Data Structures

Self-study: Some New Features in C++11

Dr. Desmond Tsoi

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

A List of New Features in C++11

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

Part I

Uniform and General Initialization Using { }-Lists and Prevention of Narrowing

{ }

= and { } Initializer for Variables

- 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};        // But = 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

Part II

Generalized and Guaranteed Constant Expressions: constexpr

CONSTEXPR

constexpr

- constexpr is a construct in C++11 to improve the performance of programs by doing computations at compile time rather than runtime.
- It specifies that the value of an object or a function can be evaluated at compile time and the expression can be used in other constant expressions.
- Restrictions of constexpr function
  1. In C++11, a constexpr function should contain only ONE return statement. (Relaxed in C++14)
  2. Each of its parameters must be a literal type.
  3. Its return type should not be void type and other operator like prefix increment are not allowed in constexpr function. It must be a literal type (e.g. scalar type, reference type, an array of literal type).
  4. A constexpr function should refer only constant global variables.
  5. A constexpr function can call only other constexpr functions.
  6. A constexpr function has to be non-virtual.

Constant Expression Example 1

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

constexpr int addition(int x, int y)
{
  return (x + y);
}

int main()
{
  const int sum = addition(10, 20);  // Evaluate at compile time
  cout << sum << endl;
  return 0;
}
```

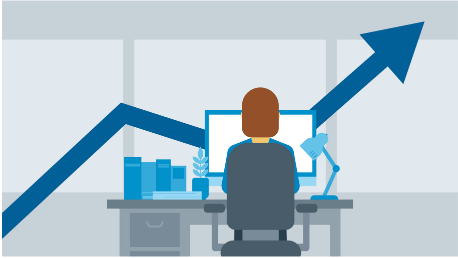
Constant Expression Example 2 (More Than One Return Statements)

```
1   #include <iostream>    /* File : constexpr-find-max.cpp */
2   using namespace std;
3
4   constexpr int find_max(int x, int y)
5   {
6     if(x > y)
7       return x;
8     else
9       return y;
10  }
11
12  int main()
13  {
14    int max = find_max(20, 30);
15    cout << max << endl;
16    return 0;
17  }
```

```
constexpr-find-max.cpp: In function âconstexpr int find_max(int, int):
constexpr-find-max.cpp:10:1: error: body of constexpr function
constexpr int find_max(int, int) not a return-statement
 }
 ^
```

Constant Expression Example 2 (Updated)

```
#include <iostream>    /* File : constexpr-find_max2.cpp */
using namespace std;

constexpr int find_max(int x, int y)
{
  return (x > y) ? x : y;
}

int main()
{
  int max = find_max(20, 30);
  cout << max << endl;
  return 0;
}
```



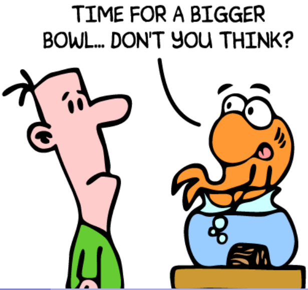
Constant Expression Example 3 (Access Non-const Global Variable)

```
#include <iostream>    /* File : constexpr-bigger-than.cpp */
using namespace std;

int ten = 10;

constexpr bool bigger_than(int x) { return x > ten; }

int main() {
  if(bigger_than(21))
    cout << "21 is bigger than 10" << endl;
  else
    cout << "21 is not bigger than 10" << endl;
  return 0;
}
```

```
constexpr-bigger-than.cpp: In function âconstexpr bool bigger_than(int)â:
constexpr-bigger-than.cpp:8:1: error: the value of "ten" is not usable in a
constant expression
 }
 ^
constexpr-bigger-than.cpp:4:5: note: "int ten" is not const
 int ten = 10;
     ^
```

Constant Expression Example 3 (Updated)

```
#include <iostream>    /* File : constexpr-bigger-than2.cpp */
using namespace std;

const int TEN = 10;

constexpr bool bigger_than(int x) { return x > TEN; }

int main() {
  if(bigger_than(21))
    cout << "21 is bigger than 10" << endl;
  else
    cout << "21 is not bigger than 10" << endl;
  return 0;
}
```



Constant Expression Example 4 (Calling Non-constexpr Function)

```
#include <iostream>    /* File : constexpr-prime-bigger-than.cpp */
using namespace std;

const int TEN = 10;

bool is_prime_recursive(int x, int c) {
  return (c*c > x) ? true : (x % c == 0) ? false : is_prime_recursive(x, c+1);
}

bool is_prime(int x) { return (x <= 1) ? false : is_prime_recursive(x, 2); }

constexpr bool prime_bigger_than(int x) { return is_prime(x) && x > TEN; }

int main() {
  if(prime_bigger_than(13))
    cout << "13 is a prime number and bigger than 10" << endl;
  else
    cout << "13 is either not a prime number or smaller than 10" << endl;
  return 0;
}
```

```
constexpr-prime-bigger-than.cpp: In function âconstexpr bool prime_bigger_than(int):
constexpr-prime-bigger-than.cpp:12:60: error: call to non-constexpr
function bool is_prime(int) constexpr
bool prime_bigger_than(int x) { return is_prime(x) && x > TEN; }
```

Constant Expression Example 4 (Updated)

```
#include <iostream>     /* File : constexpr-prime-bigger-than2.cpp */
using namespace std;

const int TEN = 10;

constexpr bool is_prime_recursive(int x, int c) {
  return (c*c > x) ? true : (x % c == 0) ? false : is_prime_recursive(x, c+1);
}

constexpr bool is_prime(int x) {
  return (x <= 1) ? false : is_prime_recursive(x, 2);
}

constexpr bool prime_bigger_than(int x) { return is_prime(x) && x > TEN; }

int main() {
  if(prime_bigger_than(13))
    cout << "13 is a prime number and bigger than 10" << endl;
  else
    cout << "13 is either not a prime number or smaller than 10" << endl;
  return 0;
}
```

constexpr with Constructors and Objects

- constexpr can be used in constructors and objects.

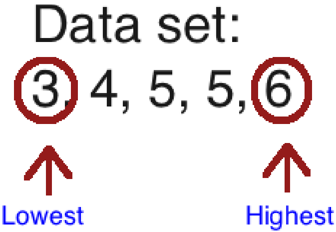
```
#include <iostream>    /* File : constexpr-constructor-object.cpp */
using namespace std;

class Rectangle {
  private:
    int width {0};
    int height {0};

  public:
    // A constexpr constructor
    constexpr Rectangle(int width, int height) : width(width), height(height) {}
    constexpr int getArea() { return width * height; }
};

int main() {
  // rect is initialized at compile time
  constexpr Rectangle rect(10, 20);
  cout << rect.getArea();
  return 0;
}
```

constexpr vs. inline Functions

- Both constexpr and inline functions are for performance improvements.
- inline functions request compiler to expand at compile time and save time of function call overheads.
- Expressions in inline functions are always evaluated at runtime, but expressions in constexpr function are evaluated at compile time.

Part III

Range-for-Statement



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;
}
```

Program Output of Range-for Example

```
Square some numbers in a list
0
1
4
9
16
Square the numbers in range
4
25
729
1600
Print the numbers in range
2
5
27
40
Again print the numbers in range
4
10
54
80
```

Part IV

Local Anonymous Functions — Lambdas

Lambda Expressions (Lambdas)

Syntax: Lambda

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

- They are anonymous function — 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 the 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. In fact, it is an error to capture them in a lambda.
- 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;
}
```

Program Output of Simple Lambdas with No Captures

```
4
25
49
100
4          10         14         20
6
9
7
```

Example: Lambdas with Captures

```
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!

Program Output

```
3
6
11
18
a = 1   b = 2   c = 3
3
6
11
18
a = 11  b = 12  c = 13
```

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?

```
a = 2   b = 5   c = 7
a = 22  b = 15  c = 37
```

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 [&].

Program Output of Mutable Lambda with Return Example

```
a = 1.6 b = 3.7 c = 5.3 f(1) = 5
a = 1.6 b = 5.7 c = 8.9 f(2) = 8
a = 1.6 b = 5.7 c = 8.9
```

Example: Nested Lambda

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

int main()
{
    int a = 1, b = 1, c = 1;

    auto f = [a, &b, &c]() mutable
    {
        auto g = [a, b, &c]() mutable      // Nested lambda
        {
            cout << a << b << c << endl;
            a = b = c = 4;
        };

        a = b = c = 3; g();
    };

    a = b = c = 2; f();
    cout << a << b << c << endl;
    return 0;
}
```

Program Output:

```
123
234
```

Quiz: What if we capture `b` by value in `f` and by reference in `g`?

Part V

Delegating Constructors

Delegating Constructors

- In C++11, constructors allow to call another constructor from the same class using member initialize list syntax.
- It prevents code duplication and to delegate the initialize list.

```
#include <iostream> /* File: delegating-constructor.cpp */
#include <cstring>
using namespace std;

class Word          // Modified from copy-constructor.cpp
{
  private:
    int frequency; char* str;
  public:
    Word(const char* s, int f = 1)
    {
        frequency = f; str = new char [strlen(s)+1]; strcpy(str, s);
        cout << "conversion" << endl;
    }
    Word(const Word& w) : Word(w.str, w.frequency) { cout << "copy" << endl; }
    void print() const { cout << str << " : " << frequency << endl; }
};

int main()
{
    Word movie("Titanic"); movie.print(); // which constructor?
    Word song(movie); song.print();       // which constructor?
    Word ship = movie; ship.print();      // which constructor?
}
```

Delegating Constructors

- In this example, the copy constructor, using the member initializer list, delegates the conversion constructor, to create an object.
- Restriction: the delegated constructor (copy constructor in this case) must be the only item in the MIL.
- In fact, we can use private utility function to deal with this before C++11.

Part VI

Explicit Conversion Operators

Conversion Constructors

- Before C++11, conversion constructors can be used for explicit and implicit conversions.

```
#include <cstring>   /* File : conversion-constructor.cpp */
class Word {
  private:
    int frequency; char* str;
  public:
    Word(const char* s )
       { frequency = 1; str = new char[strlen(s) + 1]; strcpy(str, s); }
};

int main() {
  Word* p = new Word("action");     // Explicit conversion
  Word movie("Titanic");            // Explicit conversion
  Word director = "James Cameron"; // Implicit conversion
}
```

- However, a constructor is not the only mechanism for defining a conversion in C++11.
- If we cannot modify a class, we can define a conversion operator from a different class.

Conversion Operators

```
#include <iostream>    /* File: conversion-operator.cpp */
#include <cstring>
using namespace std;

class Word {
  private: int frequency; char* str;
  public:
    Word(const char* s)
      { frequency = 1; str = new char[strlen(s) + 1]; strcpy(str, s); }
};

class EnglishWord {
  private: int frequency; char* str;
  public:
    EnglishWord(const char* s)
      { frequency = 1; str = new char[strlen(s) + 1]; strcpy(str, s); }

    operator Word()
      { cout << "conversion operator is called" << endl; return Word(str); }
};

void process_word(Word aObj) {}

int main() {
  EnglishWord engWord("Titanic");
  Word word = engWord;     // Implicit conversion by surprise
  process_word(engWord);  // Implicit conversion by surprise
}
```

Explicit Conversion Operators

- Similar to constructors, explicit keyword can be added to conversion operators to prevent implicit conversion.

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

class Word {
  private: int frequency; char* str;
  public:
    Word(const char* s) { /* ... */ }
};

class EnglishWord {
  private: int frequency; char* str;
  public:
    EnglishWord(const char* s) { /* ... */ }

    explicit operator Word()
      { cout << "conversion operator is called" << endl; return Word(str); }
};

void process_word(Word aObj) { /* ... */ }

int main() {
  EnglishWord engWord("Titanic");
  Word word = engWord;     // Bug: Implicit conversion
  process_word(engWord);  // Bug: Implicit conversion
}
```

Part VII

Enum Class

enum vs enum Class

- Recall, an enumeration (enum) is a type is a user-defined type which can hold a finite set of symbolic objects.
- Limitations of enumeration.
  - Two enumeration cannot share the same identifier names.
  - No variable can be named as what is already in some enumeration.
  - Enumerations are not type safe.

Limitation 1: Cannot Share The Same Identifier Names

```cpp
#include <iostream>    /* File: enumeration-type-1.cpp */
using namespace std;

enum shapes1 { TEXT, LINE };
enum shapes2 { TEXT, LINE };

int main() {
    shapes1 shape1 = TEXT;
    shapes2 shape2 = TEXT;

    cout << shape1 << ", " << shape2 << endl;
    return 0;
}
```

```
enumeration-type-1.cpp:5:16: error: redeclaration of 'TEXT'
 enum shapes2 { TEXT, LINE };
                ^
enumeration-type-1.cpp:4:16: note: previous declaration 'shapes1 TEXT'
 enum shapes1 { TEXT, LINE };
                ^
enumeration-type-1.cpp:5:22: error: redeclaration of 'LINE'
 enum shapes2 { TEXT, LINE };
                      ^
enumeration-type-1.cpp:4:22: note: previous declaration 'shapes1 LINE'
 enum shapes1 { TEXT, LINE };
                      ^
```

Limitation 2: No Variable Can Have A Name Which Is Already in Some Enumeration

```cpp
#include <iostream>    /* File: enumeration-type-2.cpp */
using namespace std;

int main() {
    enum shapes { TEXT, LINE };

    shapes shape = TEXT;
    const int TEXT = 88;

    cout << shape << endl;
    return 0;
}
```

```
enumeration-type-2.cpp: In function 'int main()':
enumeration-type-2.cpp:8:12: error: 'const int TEXT' redeclared as different kind
of symbol
  const int TEXT = 88;
            ^
enumeration-type-2.cpp:5:16: error: previous declaration of 'main()::shapes TEXT'
  enum shapes { TEXT, LINE };
                ^
```

Limitation 3: Enumerations Are Not Type Safe

```cpp
#include <iostream>    /* File: enumeration-type-3.cpp */
using namespace std;

enum shapes { TEXT, LINE };
enum color { RED, GREEN, BLUE };

int main() {
    shapes shape = TEXT;
    color color = RED;

    if(shape == color)
        cout << "Equal" << endl;
    return 0;
}
```

```
enumeration-type-3.cpp: In function 'int main()':
enumeration-type-3.cpp:11:14: warning: comparison between âenum shapesâ and
'enum color' [-Wenum-compare]
  if(shape == color)
             ^
```

Enum Class

- C++11 introduces enum classes (also called scoped enumerations), that makes enumerations both strongly typed and strongly scoped.
- Class enum does not allow implicit conversion to int, and also does not compare enumerators from different enumerations.

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

int main() {
    enum class color1 { RED, GREEN, BLUE };
    enum class color2 { RED, BLACK, WHITE };
    enum class shapes { TEXT, LINE };

    const int RED = 10; // OK, different scope
    color1 x = color1::GREEN;

    // Type safe
    cout << (x == color1::RED) ? "It is Red\n" : "It is not Red\n";

    shapes p = shapes::TEXT;
    if(x == p)     // Error
        cout << "GREEN is equal to TEXT";

    cout << x << endl;     // Error
    cout << (int)x << endl;     // OK
    return 0;
}
```

That's all!

Any questions?