W.P.Hwang
CS-CCS-NCTU

C++11 supplementary

Lambda expression

- A lambda expression denotes an anonymous function.
- Basic syntax

  [*capture***]** **(***parameters***)** **->** *return-type* **{** *body* **}**

  Example

  ```
  [](int x) -> int { return x*x; }
  ```

  In case the trailing return type is omitted, the type of **`x*x`** is the return type.
- A lambda expression creates a function object of a unique class type – called the *closure type* – that supports **`operator()`**.
- Example

  ```
  int main()
  {
     cout << [](int x){ return x*x; }(3);
     cout << [](int x){ return x*x; }.operator()(3);
  }
  ```
- The name of the closure type of each lambda expression is uniquely generated by the compiler.
  E.g. the two lambda expressions in preceding example are of distinct type.
- To give a lambda expression a name, the name of its closure type must be known. To this end, we may resort to **`auto`**, **`decltype`**, template argument deduction, etc.

- Example

```
int main()
{
    auto f = [](int x){ return x*x; }
    decltype(f) g = f;
    cout << f(3) << g(3);
    int a[7]={1,2,3,4,5,6,7}; // C++ as a better C, p63
    cout << accumulate(a,a+7,0,
                 [](int x,int y){ return x+y; });
}
```

Use template argument deduction to deduce its type.

- A lambda expression with free variables is meaningless.
  For example, what is the meaning of this lambda expression?

```
[](int x){ return x+y; }
```

y is a free variable

- Free variables must be captured by value (copy) or reference.
- Example

may be omitted

```
int main()
{
    int x=2,y=3;
    auto f = [x,y](){ return x+y; };     // value
    auto g = [&x,&y](){ return x+y; };   // reference
    x=4; y=5;
    cout << f() << g();      // 59
}
```

Comments

```
[=]{ return x+y; };      // default capture by value
[&]{ return x+y; };      // default capture by reference
```

// both capture **x** by value and **y** by reference

```
[=,&y]{ return x+y; }
[&,x]{ return x+y; }
```

- Example

```
#include <algorithm>        // for  for_each
int main()
{
   int a[7]={1,2,3,4,5,6,7};
   int sum=0;
   for_each(a,a+7,[&sum](int x)->void{ sum+=x; });
   cout << sum;
}                                   // may be omitted
```

Note that the call to **`for_each`** essentially executes the loop:

```
for (int* it=a;it!=a+7;++it)
   [&sum](int x){ sum+=x; }(*it);
```

- Example (May be skipped on first reading)

```
int main()
{
   int x=2,y=3;
   auto f = [x,&y]{ return x+y; };
   x=4; y=5;
   cout << f();
}
```

is compiled to something like

```
int main()
{
   int x=2,y=3;
   class I_have_no_name {
   public:
      I_have_no_name(int a,int& b) : x(a),y(b) {}
      int operator()() const { return x+y; }
   private:
      int x,&y;
   };
   auto f = I_have_no_name(x,y);
   x=4; y=5;
   cout << f();
}
```

Polymorphic function wrapper

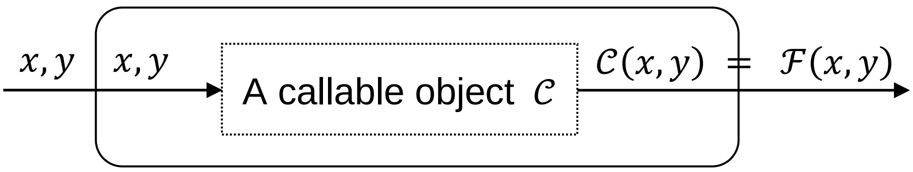
- The **`function`** class template provides polymorphic wrappers that encapsulate arbitrary callable objects.
- Example

The type
`std::function<int(int,int)>`
encapsulates all callable objects that have the call signature **`int(int,int)`**.

```
#include <functional>
int add(int x,int y) { return x+y; }
int main()
{
   typedef function<int(int,int)> bfn;
   bfn f = [](int x,int y){ return x+y; };
   bfn g[2] = {plus<int>(),add};
   cout << f(2,3) << g[0](2,3) << g[1](2,3);
}
```

Comment

A **`bfn`** object holds a callable object and supports a call operation that forwards to that object.



`bfn` object $\mathcal{F}$

```
int bfn::operator()(int x,int y) const
{
   return 𝒞(x,y);          // ℱ forwards x and y to 𝒞
}
```

Notice that , for $\mathcal{C} =$ **`plus<int>()`**, $\mathcal{F}$ forwards **`x`** and **`y`** to $\mathcal{C}$ by reference. (This is OK.)
For the other two cases, $\mathcal{F}$ forwards **`x`** and **`y`** to $\mathcal{C}$ by value.

- Example – Function composition; C++ as a better C, p65

  // Version A

```
function<int(int)> c(int f(int),int g(int))
{
   return [f,g](int x){ return f(g(x)); };
}
int f(int x) { return x+x; }
int g(int x) { return x*x; }
int main()
{
  cout << c(f,g)(3) << endl;
}
```

  // Version B – File comp.cpp

```
#include <iostream>
#include <functional>
using namespace std;
typedef function<int(int)> ufn;
ufn c(ufn f,ufn g)
{
   return [f,g](int x){ return f(g(x)); };
}
int main()
{
   cout << c([](int x){ return x+x; },
                 [](int x){ return x*x; })(3);
   cout << endl;
}
```

  Note: Use GNU C++ compiler to compile the file comp.cpp.

  bsd2> g++47 -std=c++11 -rpath=/usr/local/lib/gcc47 comp.cpp
  bsd2> ./a.out
  18 for GLIBCXX_3.4.14

auto specifier

- **`auto`** is no longer is storage class specifier, e.g.

```
void p()
{
   auto int x;          // error in C++11
   static int y;
}
```

- **`auto`** is now a type specifier, signifying that
  1. the type of a variable being declared shall be deduced from its initializer using template argument deduction, or
  2. a function declarator shall include a *trailing-return-type*.

- Example

```
void p()
{
   auto x=3;              // x has type int
   const auto* y=&x;      // y has type const int*
   static auto z=x;       // z has type int
}
```

Trailing return type

- Trailing-return-types are convenient when the return type of a function is complex.

- Example (C++ as a better C - p.65)

```
auto msg() -> void { cout << "hello\n"; }
auto mkmsg() -> void (*)() { return msg; }
auto main() -> int {  mkmsg()(); (*mkmsg())(); }
```

List-initialization

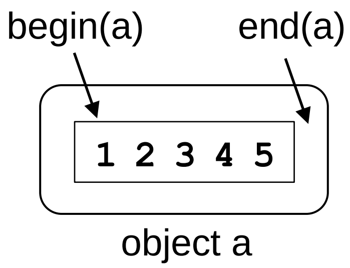
- List-initialization is the initialization of an object from a braced initializer list.
- Narrowing conversions are not allowed at the top level in list-initializations.
- Example

```
// variable initialization
int a[2]={1,2};               // ok, as usual
int b[2]={1,2.0};             // error in C++11, narrowing
int c[2]={1,(int)2.0};        // ok, not a top-level narrowing
int d[2]{1,2};                // new in C++11
int e[2]{};                   // default to 0,0

struct X { int x,y; };
X a={1,2};
X d{1,2};
X f({1,2});
// Only class type can parenthesize a braced initializer list

int a={2};
int d{2};
// Q: Which is ill-formed?
// int x={2.0},y{2.0},z=2.0,w(2.0);

// assignment
d={3};

// new expression
int* a=new int{2};
int* b=new int[3]{1,2,3};
X* c=new X[3]{{1,2},{3,4},{5,6}};
int n=2;
int* d=new int[n]{1,2,3};
// Warning, unable to verify the length of initializer list
int* e=new (operator new(sizeof(int))) int{2};
```

- Example (Cont'd)

// return statement
```
#include <utility>
pair<int,int>  f() { return {1,2}; }
```

// function argument
```
#include <initializer_list>
int sum(initializer_list<int> a)
{
   int s=0;
   for (const int* it=begin(a);it!=end(a);++it)
      s+=*it;
   return s;
}
int main()
{
   cout << sum({1,2,3,4,5});
}
```



Comment

An object of type `initializer_list<T>` provides access to an array of objects of type `const T`.

Range-based `for` statement

- Syntax

`for` **`(`** *for-range-declaration* **:** *expression* **`)`** *statement*
`for` **`(`** *for-range-declaration* **:** *braced-init-list* **`)`** *statement*

- Example

The preceding `for` loop may be written as:
```
for (int i : a) s+=i;
```

```
int array[5] = {1,2,3,4,5};
for (int& i : array) i++;
for (int i : {1,2,3,4,5}) cout << i;
for (char c : "Snoopy") cout << c;
```