c++11 new features

constexptr

overview

- `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***.
- could be an expression our value.

Restrictions

- `constexptr`function should only have **one** return statement
- Each of its **parameters** must be a **literal** type.
- return type should not be `void` type and other `operator` like prefix increment are not allowed in `constexpr` function.
- A `constexpr` function should refer only **constant global** variables.

> *can call **only** other `constexpr` functions.*
>
> *has to be **non-virtual**.*

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

Range

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

```
for (int k : { 0, 1, 2, 3, 4 })
for (<statement> : <range>)
```

Lambdas -*local anonymous functions*

syntax

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

details

- They are usually defined **locally** inside functions, though **global** lambdas are also possible.

1. capture - [ ]

- 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`.*

- **empty capture** `[ ]`
  indicates that the body of the `lambda` expression accesses no variables in the enclosing scope.

- `[=]`:
  capture all local variables by **value**.

- `[&]`:
  capture all local variables by **reference**.

Capture by Value or Reference

1. When a lambda expression captures variables by **value**, the values are captured by copying **only once** at the time the lambda is defined.
2. **Capture-by-value** is similar to **pass-by-value**.
3. Unlike **PBV**, variables captured by value **cannot** be modified **inside** the lambda unless you make it `mutable`.
4. Similarly, **capture-by-reference** is similar to **pass-by-reference**.
   example:

```
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!

```

examples

Simple Lambdas with No Captures

```
int range[] = { 2, 5, 7, 10 };
for (int v : range)
{
cout << [](int k) { return k * k; } (v) << endl;
}
```

```
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;
```

Lambdas with Captures

```
int sum = 0, a = 1, b = 2, c = 3;
cout << [=](int x) { return a*x*x + b*x + c; } (k) << endl;
cout << [&](int x) { a += x*x; return b += x; } (k) << endl;
```

`auto` expression

the `auto` expression enables us to declare a variable without a type which will automatically inferred by the compiler.
for example:

- Universal initialization syntax, such as `auto a { 42 };`.
- Assignment syntax, such as `auto b = 0;`.
- Universal assignment syntax, which combines the two previous forms, such as `auto c = { 3.14156 };`.
- Direct initialization, or constructor-style syntax, such as `auto d( 1.41421f );`
-

```
auto y = [] (auto first, auto second) { return first + second; };
```

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.

difference between (=)and { })

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

class member function

inline function

For inline function, the compiler will replace the function body when the function is called by copying and pasting.

An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. This eliminates call-linkage overhead and can expose significant optimization opportunities.

Using the `inline` specifier is only a suggestion to the compiler that an inline expansion can be performed; the compiler is **free to ignore the suggestion**.

After compiling, the inline function will be no longer be a function when function call.

> *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; }
};

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

```

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.

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

```
int height =10;
class Weird
{
    short height;
    Weird() { height = 5; }
    // to access the global varible height,
    // we need to use ::height
}
```

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;
}
// Person* this will autonmatically add as an argument.
```

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