OOP

Gavrilut Dragos

Course 9

Summary

- Lambda expressions
- Modeling lambda expression behavior
- Implicit conversion to a pointer to a function
- Lambdas and STL
- Using lambda with templates (Generic lambdas)
- Mutable capture
- Initialized lambda capture
- New feature in C++17 and beyond

Lambda expressions

Lambda expressions

- Lets assume that we have the following structure:

App.cpp

```
struct Student
{
	const char * Name;
	int Grade;
	int Group;
	int Age;
};

Student students[] = {
	{ "Popescu",  10, 5, 19 },
	{ "Ionescu",   8, 3, 20 },
	{ "Georgescu", 9, 4, 21 },
};
```

- ‘students’ is a global list of students.

Lambda expressions

- Now let's assume that we want to sort the entire student list in different ways (alphabetically, after the 'Age' field, based on the 'Grade' field, etc).
- The easiest way would be to create a sort algorithm that uses a pointer to a function used to compare to Student structures.

App.cpp

```
void Sort(Student *list, int count, bool(*BiggerFnc)(Student &s1, Student &s2) ) {
	bool sorted;
	do {
		sorted = true;
		for (int tr = 0; tr < count - 1; tr++)
		{
			if (BiggerFnc(list[tr], list[tr + 1]))
			{
				Student aux = list[tr];
				list[tr] = list[tr + 1];
				list[tr + 1] = aux;
				sorted = false;
			}
		}
	} while (!sorted);
}
```

Lambda expressions

- The main function will look as follows:

App.cpp

```
struct Student { … }
Student students[3] = { … }
void Sort(Student *list, int count, bool(*BiggerFnc)(Student &s1, Student &s2) ) { … }

bool ByGrade (Student &s1, Student &s2)
{
      return s1.Grade > s2.Grade;
}
bool ByAge (Student &s1, Student &s2)
{
      return s1.Age > s2.Age;
}
bool ByName (Student &s1, Student &s2)
{
      return strcmp(s1.Name, s2.Name) > 0;
}

int main() {
      Sort(students, 3, ByGrade );
      Sort(students, 3, ByAge );
      Sort(students, 3, ByName );
}
```

Lambda expressions

- Instead of creating functions for each comparation, wouldn’t it be easier if we could just write the comparation whenever we call the Sort function.

App.cpp

```
struct Student { … }
Student students[3] = { … }
void Sort(Student *list, int count, bool(*BiggerFnc)(Student &s1, Student &s2) ) { … }


int main() {
      Sort(students, 3, [](Student &s1, Student &s2) { return s1.Grade > s2.Grade; } );
}
```

- The code compiles and works as expected, the list of students being sorted based on the ‘Grade’ field.

Lambda expressions

- A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- Let’s analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
      return lambda x: x * factor

def main():
      f1 = Multiply(5)
      f2 = Multiply(7)
      print f1(3),f2(3)

main()
```

Lambda expressions

- A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
      return lambda x: x * factor

def main():
      f1 = Multiply(5)
      f2 = Multiply(7)
      print f1(3),f2(3)

main()
```

Multiply function actually returns another function (that has a prototipe like the following:)

def Function(x): returns x * factor

Lambda expressions

- A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
      return lambda x: x * factor

def main():
      f1 = Multiply(5)
      f2 = Multiply(7)
      print f1(3),f2(3)

main()
```

f1 will be a function that has the following prototype:

def Function(x): returns x * 5

The *factor* parameter from the Multiply function is used in this function (as the value 5).

Lambda expressions

- A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
      return lambda x: x * factor

def main():
      f1 = Multiply(5)
      f2 = Multiply(7)
      print f1(3),f2(3)

main()
```

f1 will be a function that has the following prototype:

def Function(x): returns x * 7

The *factor* parameter from the Multiply function is used in this function (as the value 7).

Lambda expressions

- A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- Let’s analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
      return lambda x: x * factor

def main():
      f1 = Multiply(5)
      f2 = Multiply(7)
      print f1(3),f2(3)

main()
```

The code runs and prints 15 (3 x 5) and 21 (3 x 7)

Lambda expressions

- A lambda expression is defined in the following ways:

Lambda expressions
`[captures] (parameters) -> return type { body }`
`[captures] (parameters) { body }`
`[captures] { body }`

- Examples:

Lambda expressions
`[x,y] (int a, float b) -> bool { return (a*b)<(x+y); }`
`[x] (int *xx) { return *xx+x; } // the return type is deduced to be int from the body`
`[a,b] { return a+b; }`

Lambda expressions

- The *capture* component from the lambda expression can be:

Capture	Observation
`[]`	Captures nothing
`[a,b]`	Captures variables "a" and "b" by making a copy of them
`[&a,&b]`	Captures variables "a" and "b" using their reference
`[this]`	Captures current object
`[&]`	Captures all variables ***used*** in the body of the lambda by using their reference. If "this" is available it as also captured (by reference)
`[=]`	Captures all variables ***used*** in the body of the lambda by making a copy of them. If "this" is available it as also captured (by reference).
`[=,&a]`	Captures all variables ***used*** in the body of the lambda by making a copy of them, except for "*a*" that is captured by reference.
`[&,a]`	Captures all variables ***used*** in the body of the lambda by using their reference, except for "a" that is capture by making a copy.

Lambda expressions

- Example:

App.cpp

```
int main()
{
	auto f = [](int x, int y) { return x + y; };
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

- This example compiles and prints value 30 on the screen.

Lambda expressions

- Example :

App.cpp

```
int main()
{
	int value = 100;
	auto f = [value] (int x, int y) { return x + y + value; };
	printf("%d\n", f(10, 20));
	value = 200;
	printf("%d\n", f(10, 20));
	return 0;
}
```

- This example compiles and prints value 130 twice on the screen.

Lambda expressions

- Example :

App.cpp

Capture local variable 'value' by making a copy

```
int main()
{
      int value = 100;
      auto f = [value] (int x, int y) { return x + y + value; };
      printf("%d\n", f(10, 20));
      value = 200;
      printf("%d\n", f(10, 20));
      return 0;
}
```

- This example compiles and prints value 130 twice on the screen.
- Local variable 'value' is capture by making a copy of its value. This means that event if we change its value the result from the lambda function will be the same.

Lambda expressions

- Example :

App.cpp

```
int main()
{
	int value = 100;
	auto f = [&value] (int x, int y) { return x + y + value; };
	printf("%d\n", f(10, 20));
	value = 200;
	printf("%d\n", f(10, 20));
	return 0;
}
```

Capture local variable ‘value’ by reference

- Now the code runs and prints 130 and them 230 on the screen.

Lambda expressions

- Example :

App.cpp

```
int main()
{
	int value = 100;
	auto f = [&value] (int x, int y) -> char { return x + y + value; };
	printf("%d\n", f(10, 20));
	value = 200;
	printf("%d\n", f(10, 20));
	return 0;
}
```

Lambda expression type

- In this case we set up the type of the lambda expression. If not set it is deduced from the return type of the lambda expression.
- The result will be -126 and -30 (char representation for int values 130 and 230)

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [=] (int x, int y)  { return x + y + a + b + c + aa; };
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- The code compiles and prints the value 1060.
- All local variables and parameters from the function "MyFunction" are captured.

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int
{
      int a, b, c;
      a = b = c = 10;
      auto f = [=] (int x, int y)  { return x + y + a + b + c + aa; };
      printf("%d\n", f(10, 20));
}
int main()
{
      MyFunction(1000);
      return 0;
}
```

Capture all local variables and parameters by making a copy of them

- The code compiles and prints the value 1060.
- All local variables and parameters from the function “MyFunction” are captured.

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [&] (int x, int y)  { return x + y + a + b + c + aa; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- The code compiles and prints the value 1060 and then 2330
- All local variables and parameters from the function “MyFunction” are captured.

Lambda expressions

- Example :

Capture all local variables and parameters by reference

App.cpp

```
void MyFunction(int
{
	int a, b, c;
	a = b = c = 10;
	auto f = [&] (int x, int y)  { return x + y + a + b + c + aa; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- The code compiles and prints the value 1060 and then 2330
- All local variables and parameters from the function “MyFunction” are captured (by reference).

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [&, a] (int x, int y)  { return x + y + a + b + c + aa; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- The code compiles and prints the value 1060 and then 2240. All variables a capture by reference except for local variable “a” that is capture by making a copy of itself.

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [&, &a] (int x, int y)  { return x + y + a + b + c + aa; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- Depending on the compiler this code might work. “Cl.exe” does not compile, GCC compiles with an warning.

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa
{
      int a, b, c;
      a = b = c = 10;
      auto f = [&, &a] (int x, int y)  { return x + y + a + b + c + aa; };
      printf("%d\n", f(10, 20));
      a = b = c = 100;
      aa *= 2;
      printf("%d\n", f(10, 20));
}
int main()
{
      MyFunction(1000);
      return 0;
}
```

> **GCC:** warning: explicit by-reference capture of ‘a’ redundant with by-reference capture default
> **CL:** error C3488: '&a' cannot be explicitly captured when the default capture mode is by reference (&)

- Depending on the compiler this code might work. “Cl.exe” does not compile, GCC compiles with an warning.

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [=, &a] (int x, int y)  { return x + y + a + b + c + aa; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- The code compiles and prints the value 1060 and then 1150. All variables a capture by making a copy of themselves except for local variable “a” that is capture by reference.

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [=, a] (int x, int y)  { return x + y + a + b + c + aa; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- Depending on the compiler this code might work. “Cl.exe” does not compile, GCC compiles with an warning.

Lambda expressions

- Example :

App.cpp

error C3483: 'a' is already part of the lambda capture list

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [a, a] (int x, int y)  { return x + y + a; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

- This code will not compile as local variable ‘a’ can not be capture twice.

Lambda expressions

- Example :

App.cpp

```
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto f = [a, &a] (int x, int y)  { return x + y + a; };
	printf("%d\n", f(10, 20));
	a = b = c = 100;
	aa *= 2;
	printf("%d\n", f(10, 20));
}
int main()
{
	MyFunction(1000);
	return 0;
}
```

error C3483: 'a' is already part of the lambda capture list

- This code will not compile as local variable ‘a’ can not be capture twice. In this case we tried to capture ‘a’ making a copy of itself and also by reference.

Lambda expressions

- Example :

App.cpp

```
int Add (int x, int y)
{
	return x + y;
}
void MyFunction(int aa)
{
	int a, b, c;
	a = b = c = 10;
	auto ptr_f = Add;
	auto f = [ptr_f](int x, int y)  { return ptr_f(x, y); };
	printf("%d\n", f(10, 20));
}
int main()
{	MyFunction(1000);
	return 0;
}
```

- This code compiles and prints “30” on the screen. In this case the capture variable is a pointer to a function (***Add***).

- Modeling lambda expression behavior

Lambda expressions

- Example:

App.cpp

```
int main()
{
	auto f = [](int x, int y) { return x + y; };
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

- This example compiles and prints value 30 on the screen.

Lambda expressions

- Example:

Assembly code

```
 push         ebp
 mov          ebp,esp
 sub          esp,4Ch
auto f = [](int x, int y) { return x + y; };
int x = f(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f]
 call         <lambda_1b12082d1acdf839b51735232aba4b6a>::operator()
 mov          dword ptr [x],eax

printf("X = %d", x);
 mov          eax,dword ptr [x]
 push         eax
 push         3A935Ch  // address of "X = %d" string
 call         printf
 add          esp,8

return 0;
 xor          eax,eax
```

Lambda expressions

- Example:

Assembly code

```
 push
 mov
 sub
auto f = [](i
int x = f(10,
 push
 push        0Ah
 lea         ecx,[f]
 call        <lambda_1b12082d1acdf839b51735232aba4b6a>::operator()
 mov         dword ptr [x],eax

printf("X = %d", x);
 mov         eax,dword ptr [x]
 push        eax
 push        3A935Ch  // address of "X = %d" string
 call        printf
 add         esp,8

return 0;
 xor         eax,eax
```

From the compiler point of view, lambda expressions are modeled as an object that has the () operator overwritten.

Lambda expressions

- Example:

Assembly code

```
 push        ebp
 mov         ebp,esp
 sub         esp,4Ch
auto f = [](int x, int y) { return x + y; }
int x = f(10, 20);
 push        14h
 push        0Ah
 lea         ecx,[f]
 call        <lambda_1b12082d1acdf839b51735232aba4b6a>::operator()
 mov         dword ptr [x],eax

printf("X = %d", x);
 mov         eax,dword ptr [x]
 push        eax
 push        3A935Ch  // address of "X = %d" string
 call        printf
 add         esp,8

return 0;
 xor         eax,eax
```

```
push          ebp
mov           ebp,esp
sub           esp,44h
mov           dword ptr [this],ecx

 mov           eax,dword ptr [x]
 add           eax,dword ptr [y]

mov           esp,ebp
pop           ebp
ret           8
```

Lambda expressions

- This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
	auto f = [](int x, int y) { return x + y; };
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
	int operator() (int x,int y) const { return x+y; }
	lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
};
int main()
{
	lambda_1b12082d1acdf839b51735232aba4b6a f;
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

Lambda expressions

- This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
	auto f = [](int x, int y) { return x + y; };
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
	int operator() (int x,int y) const  { return x+y; }
	lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
}
int main()
{
	lambda_1b12082d1acdf839b51735232aba4b6a f;
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

Lambda expressions

- This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
	auto f = [](int x, int y) { return x + y; };
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
	int operator() (int x,int y) const { return x+y; }
	lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
}
int main()
{
	lambda_1b12082d1acdf839b51735232aba4b6a f;
	int x = f(10, 20);
	printf("X = %d", x);
	return 0;
}
```

Lambda expressions

- This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
	auto f = [](int x, int y) { return x + y; };
	int x = f(10, 20);
	printf("SizeOf(f) = %d", sizeof(f));
	return 0;
}
```

- The results will be 1 (consistent with the fact that "f" is indeed an object of type **lambda_1b12082d1acdf839b51735232aba4b6a**

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
	int operator() (int x,int y) const { return x+y; }
	lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
}
```

Lambda expressions

- The compiler generate lambda_xxxxxxxxxxxxxx classes for each encountered lambda structure.

App.cpp

```
int main()
{
	auto f1 = [](int x, int y) { return x + y; };
	auto f2 = [](int x, int y) { return x + y; };
	int x1 = f1(10, 20);
	int x2 = f2(10, 20);
	return 0;
}
```

- In the previous case, even if according to definition, both f1 and f2 are identical, two separate classes with two separate (but identical implementation) functions that overwrite operator() will be created.

Lambda expressions

- The compiler generate lambda_xxxxxxxxxxxxx classes for each encountered lambda structure.

Assembly code

```
 push         ebp
 mov          ebp,esp
 sub          esp,50h
auto f1 = [](int x, int y) { return x + y; };
auto f2 = [](int x, int y) { return x + y; };
int x1 = f1(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f1]
 call         <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
 mov          dword ptr [x1],eax
int x2 = f2(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f2]
 call         <lambda_e213977a927692e36f5320f87e493de8>::operator() (0923280h)
 mov          dword ptr [x2],eax
return 0;
 xor          eax,eax
```

Lambda expressions

- The compiler generate lambda_xxxxxxxxxxxxx classes for each encountered lambda structure.

Assembly code

```
 push         ebp
 mov          ebp,esp
 sub          esp,50h
auto f1 = [](int x, int y) { return x + y; };
auto f2 = [](int x, int y) { return x + y; };
int x1 = f1(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f1]
 call         <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
 mov          dword ptr [x1],eax
int x2 = f2(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f2]
 call         <lambda_e213977a927692e36f5320f87e493de8>::operator() (0923280h)
 mov          dword ptr [x2],eax
return 0;
 xor          eax,eax
```

Different classes

Lambda expressions

- The compiler generate lambda_xxxxxxxxxxxxxx classes for each encountered lambda structure.

Assembly code

```
 push         ebp
 mov          ebp,esp
 sub          esp,50h
auto f1 = [](int x, int y) { return x + y; };
auto f2 = [](int x, int y) { return x + y; };
int x1 = f1(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f1]
 call         <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
 mov          dword ptr [x1],eax
int x2 = f2(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f2]
 call         <lambda_e213977a927692e36f5320f87e493de8>::operator() (0923280h)
 mov          dword ptr [x2],eax
return 0;
 xor          eax,eax
```

Different functions for operator()

Lambda expressions

- The compiler generate lambda_xxxxxxxxxxxxxx classes for each encountered lambda structure.
- This means that:
  - For every lambda construction that the programmer uses, a class will be created (it is therefor recommended that la lambda construction to be small so that they do not increase the size of the compiled program unnecessary).
  - The type of the class that uses lambda expressions is generated at the compile time → this means that whenever a lambda is used "**auto**" should be used as well.
  - The same lambda expression can be used multiple times if "**decltype**" is used (this is valid for some compilers – not all of them allow this behavior !!!)

Lambda expressions

- In the following case the usage of **decltype** allows us to reutilize the same construct multiple times

App.cpp (default constructor)

```
int main()
{
	auto f1 = [](int x, int y) { return x + y; };
	decltype(f1) f2;
	int x1 = f1(10, 20);
	int x2 = f2(10, 20);
	return 0;
}
```

App.cpp (copy constructor)

```
int main()
{
	auto f1 = [](int x, int y) { return x + y; };
	decltype(f1) f2 = f1;
	int x1 = f1(10, 20);
	int x2 = f2(10, 20);
	return 0;
}
```

- Now both “f1” and “f2” are of the same class/type.
- Default constructor does not work for every compiler (gcc does not support it, cl.exe (18.0.x.x supports it), cl.exe (19.16.27030.1 does not). The difference in this case is that the deleted constructor was not added in cl.exe (18.0.x.x)
- Copy constructor is supported by both cl (19.16.27030.1) and gcc.

Lambda expressions

- In the following case the usage of **decltype** allows us to reutilize the same construct multiple times

Assembly code

```
 push         ebp
 mov          ebp,esp
 sub          esp,50h
auto f1 = [](int x, int y) { return x + y; };
decltype(f1) f2;
int x1 = f1(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f1]
 call         <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
 mov          dword ptr [x1],eax
int x2 = f2(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f2]
 call         <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
 mov          dword ptr [x2],eax
return 0;
 xor          eax,eax
```

The same class

Lambda expressions

- Let’s analyze the following case:

App.cpp

```
int main()
{
	int a, b;
	auto f = [a,b] (int x, int y) { return x + y + a +b;  };
	int x = f(10, 20);
	printf("sizeof(f) = %d", sizeof(f));
	return 0;
}
```

- The code compiles correctly and upon execution prints to the screen value 8.
- The size changed from 1 to 8 because of the 2 variables that were captured.

Lambda expressions

- Let's analyze the following case:

Assembly code

```
push        ebp
mov         ebp,esp
sub         esp,54h
    int a, b;
    auto f = [a,b](int x, int y) { return x + y + a +b;  };
lea         eax,[b]
push        eax
lea         ecx,[a]
push        ecx
lea         ecx,[f]
call        <lambda_3c006326…>::<lambda_3c006326…> (0D928E0h)
    int x = f(10, 20);
push        14h
push        0Ah
lea         ecx,[f]
call        <lambda_3c006326…>::operator() (0D92730h)
mov         dword ptr [x],eax
    printf("sizeof(f) = %d", sizeof(f));
………
```

Lambda expressions

- Let’s analyze the following case:

Assembly code

```
 push         ebp
 mov          ebp,esp
 sub          esp,54h
     int a, b;
     auto f = [a,b](int x, int y) { return x + y +
 lea          eax,[b]
 push         eax
 lea          ecx,[a]
 push         ecx
 lea          ecx,[f]
 call         <lambda_3c006326…>::<lambda_3c006326…> (0D928E0h)
     int x = f(10, 20);
 push         14h
 push         0Ah
 lea          ecx,[f]
 call         <lambda_3c006326…>::operator() (0D92730h)
 mov          dword ptr [x],eax
     printf("sizeof(f) = %d", sizeof(f));
………
```

One difference from the previous times is that now we have a **constructor** for the lambda object

Lambda expressions

- Let's analyze the following case:

Assembly code

```
push        ebp
mov         ebp,esp
sub         esp,54h
    int a, b;
    auto f = [a,b](int x, int y) { return x + y
lea         eax,[b]
push        eax
lea         ecx,[a]
push        ecx
lea         ecx,[f]
call        <lambda_3c006326…>::<lambda_3c006326…>
    int x = f(10, 20);
push        14h
push        0Ah
lea         ecx,[f]
call        <lambda_3c006326…>::operator() (0D92730h)
mov         dword ptr [x],eax
    printf("sizeof(f) = %d", sizeof(f));
………
```

```
push        ebp
mov         ebp,esp
sub         esp,44h
mov         dword ptr [this],ecx
mov         eax,dword ptr [this]
mov         ecx,dword ptr [param_1]
mov         edx,dword ptr [ecx]
mov         dword ptr [eax],edx
mov         eax,dword ptr [this]
mov         ecx,dword ptr [param_2]
mov         edx,dword ptr [ecx]
mov         dword ptr [eax+4],edx
mov         eax,dword ptr [this]
mov         esp,ebp
pop         ebp
ret         8
```

Lambda expressions

- This means that the same code can be translated as follows:

App.cpp

```
int main() {
      int a, b;
      auto f = [a,b] (int x, int y) { return x + y + a +b;  };
      …
      return 0;
}
```

Translated code

```
class lambda_3c006326 {
      int a,b;
public:
      lambda_3c006326(int &ref_a, int &ref_b): b(ref_b), a(ref_a) {  }
      int operator() (int x,int y) const { return x + y + a + b; }
      lambda_3c006326() = delete;
}
int main() {
      int a, b;
      lambda_3c006326 f (a,b);
      …
      return 0;
}
```

Lambda expressions

- Using the references changes the code as follows:

App.cpp

```
int main() {
      int a, b;
      auto f = [a,&b] (int x, int y) { return x + y + a +b;  };
      …
      return 0;
}
```

Translated code

```
class lambda_3c006326 {
      int a;
      int &b;
public:
      lambda_3c006326(int &ref_a, int &ref_b): b(ref_b), a(ref_a) {  }
      int operator() (int x,int y) const { return x + y + a + b; }
      lambda_3c006326() = delete;
}
int main() {
      int a, b;
      lambda_3c006326 f (a,b);
      …
      return 0;
}
```

Lambda expressions

- Using **decltype** can be used for lambdas with no capture (that have a default constructor). In case of lambdas with capture **decltype** does not work.

App.cpp

```
int main() {
      int a = 10 , b = 20;
      auto f1 = [a,b] (int x, int y) { return x + y + a +b;  };

      decltype(f1) f2(a,b);

      printf("%d", f2(1, 2));
      return 0;
}
```

"f1" lambda class has a constructor with two integer parameters. However will not compile (as it is not allowed to initialize a lambda in this way).

error C3497: you cannot construct an instance of a lambda

Lambda expressions

- Using **decltype** can be used for lambdas with no capture (that have a default constructor). In case of lambdas with capture **decltype** does not work.

App.cpp

```
int main() {
      int a = 10 , b = 20;
      auto f1 = [a,b] (int x, int y) { return x + y + a +b;  };

      decltype(f1) f2 = f1;

      printf("%d", f2(1, 2));
      return 0;
}
```

- This code will compile – a copy constructor between "f1" and "f2" is called.
- The code works and prints 33 into the screen.

Lambda expressions

- Using **decltype** can be used for lambdas with no capture (that have a default constructor). In case of lambdas with capture **decltype** does not work.

App.cpp

```
int main() {
      int a = 10 , b = 20;
      auto f1 = [a,b] (int x, int y) { return x + y + a +b;  };

      decltype(f1) f2 = {1,2};

      printf("%d", f2(1, 2));
      return 0;
}
```

- This code works on cl.exe (19.16.27030.1) for Windows but does not work on gcc
- Because of the initializer list “f2” in instantiated with two different values for internal (captured) “a” and “b”. On cl.exe for Windows the code works and prints 6.

Lambda expressions

- Copy constructor is used whenever the capture is done based on the value.

App.cpp

```
class MyNumber {
public:
	int a, b;
	MyNumber(int x,int y): a(x), b(y) { }
	MyNumber(const MyNumber &m) { a = m.b; b = m.a; }
};
int main() {
	MyNumber m(2, 3);
	auto f = [m](int x, int y) { return x * m.a + y * m.b;  };
	printf("%d\n", f(10, 20));
	return 0;
}
```

- In this case, when object “f” is created , the copy constructor for MyNumber is called and the actual object that is created within the lambda object has fields “a” and “b” reversed.
- The result of this code will be: x (10) * m.a (3) + y (20) * m.b (2) = 70

Lambda expressions

- However, if using references the copy constructor is not called and the result will be different.

App.cpp

```
class MyNumber {
public:
      int a, b;
      MyNumber(int x,int y): a(x), b(y) { }
      MyNumber(const MyNumber &m) { a = m.b; b = m.a; }
};
int main() {
      MyNumber m(2, 3);
      auto f = [&m](int x, int y) { return x * m.a + y * m.b;  };
      printf("%d\n", f(10, 20));
      return 0;
}
```

- The result of this code will be: x (10) * m.a (2) + y (20) * m.b (3) = 80

Lambda expressions

- Using “=“ and/or “&” means that only values used in the body of the lambda are actually used (copied/referenced) in la lambda class.

App.cpp

```
int main(){
	int a1, a2, a3, a4, a5, a6;
	auto f = [=] (int x, int y) { return x + y + a1 + a3;  };
	printf("%d\n", sizeof(f));
	return 0;
}
```

- The result is 8 (only a1 and a3 are copied).

App.cpp

```
int main(){
	int a1, a2, a3, a4, a5, a6;
	auto f = [=] (int x, int y) { return x + y + a1 + a3 + a5;  };
	printf("%d\n", sizeof(f));
	return 0;
}
```

- Now the result is 12 (a1, a3 and a5 are used)

Lambda expressions

- Lambdas can be used with classes and can capture **this** pointer

App.cpp

```
class Student{
public:
	const char *Name;
	int Grade;
public:
	Student(const char *n, int g) { Name = n; Grade = g; }
	void IncrementGrade()
	{
		auto la = [this] () { this->Grade++; };
		la();
	}
};
int main(){
	Student s("Popescu", 8);
	s.IncrementGrade();
}
```

- After the call of ***s.IncrementGrade*** the value of ***s.Grade*** will be 9

Lambda expressions

- Lambdas can be used with classes and can capture **this** pointer

App.cpp

```
class Student{
private:
	const char *Name;
	int Grade;
public:
	Student(const char *n, int g) { Name = n; Grade = g; }
	void IncrementGrade()
	{
		auto la = [this]() { this->Grade++; };
		la();
	}
};
int main(){
	Student s("Popescu", 8);
	s.IncrementGrade();
}
```

- Keep in mind that lambdas work similar to a friend function. Event if data members are private they can still be accessed. This code will run and the value of field Grade will be incremented.

Lambda expressions

- Lambdas can be used with classes and can capture **this** pointer

App.cpp

```
class Student{
private:
	const char *Name;
	int Grade;
public:
	Student(const char *n, int g) { Name = n; Grade = g; }
	void IncrementGrade()
	{
		auto la = [=] () { this->Grade++; };
		la();
	}
};
int main(){
	Student s("Popescu", 8);
	s.IncrementGrade();
}
```

- The same happens if we capture **this** by using '='

Lambda expressions

- Lambdas can be used with classes and can capture **this** pointer

App.cpp

```
class Student{
private:
	const char *Name;
	int Grade;
public:
	Student(const char *n, int g) { Name = n; Grade = g; }
	void IncrementGrade()
	{
		auto la = [&] () { Grade++; };
		la();
	}
};
int main(){
	Student s("Popescu", 8);
	s.IncrementGrade();
}
```

- The same happens if we capture **this** by using '&'. Also the use of "**this->**" pointer in lambda function is not required.

Lambda expressions

- Lambdas can be used with classes and can capture **this** pointer

App.cpp

```
class Student{
private:
	const char *Name;
	int Grade;
public:
	Student(const char *n, int g) { Name = n; Grade = g; }
	void IncrementGrade()
	{
		auto la = [] () { this->Grade++; };
		la();
	}
};
int main(){
	Student s("Popescu", 8);
	s.IncrementGrade();
}
```

- However, this code will not work as **this** pointer is not captured.

- Implicit conversion to a pointer to a function

Lambda expressions

- All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that “this” pointer for the lambda structure is unnecessary.
- The following code works because “f” has no capture.

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { … }
int main()
{
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	auto f = [](int n1, int n2) { return n1 > n2; };
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

Lambda expressions

- All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that “this” pointer for the lambda structure is unnecessary.
- The following code will not work as “f” captures local variable “a”

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { … }
int main()
{
	int a = 100;
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	auto f = [a] (int n1, int n2) { return n1 > n2; };
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

Lambda expressions

- All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that "this" pointer for the lambda structure is unnecessary.
- The following code will work. "f" captures all used local variables / parameters by making a copy of them, but as the body of the lambda does not use any of them, the lambda is actually without capture.

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { … }
int main()
{
	int a = 100;
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	auto f = [=] (int n1, int n2) { return n1 > n2; };
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

Lambda expressions

- All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that “this” pointer for the lambda structure is unnecessary.
- The following code will NOT work because “f” captures all local variables/parameters by value (making a copy of them) and the body actually uses one of them (“a”).

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { … }
int main()
{
	int a = 100;
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	auto f = [=] (int n1, int n2) { return n1 > (n2 + a ); };
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

Lambda expressions

- How does the compiler models this behavior

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { … }
int main()
{
      int numbers[] = { 1, 4, 2, 6, 1, 3 };
      auto f = [=] (int n1, int n2) { return n1 > n2 ; };
      bool res = f(1, 2);
      Sort(numbers, sizeof(numbers) / sizeof(int), f);
      return 0;
}
```

Lambda expressions

- How does the compiler models this behavior

Assembly code

```
    bool res = f(1, 2);
push         2
push         1
lea          ecx,[f]
call         <lambda_d87c98bc2…>::operator() (02929E0h)
mov          byte ptr [res],al
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
lea          ecx,[f]
call         <lambda_d87c98bc2…>::operator bool (__cdecl*)(int,int) (0293270h)
push         eax
push         6
lea          eax,[numbers]
push         eax
call         Sort (02912F3h)
add          esp,0Ch
    return 0;
xor          eax,eax
```

Lambda expressions

- How does the compiler models this behavior

Assembly code

```
    bool res = f(1, 2);
push          2
push          1
lea           ecx,[f]
call          <lambda_d87c98bc2…>::operator()
mov           byte ptr [res],al
    Sort(numbers, sizeof(numbers) / sizeof(int),
lea           ecx,[f]
call          <lambda_d87c98bc2…>::operator bool (__cdecl*)(int,int) (0293270h)
push          eax
push          6
lea           eax,[numbers]
push          eax
call          Sort (02912F3h)
add           esp,0Ch
    return 0;
xor           eax,eax
```

The compiler creates a cast function (that can cast to a pointer to a function).

Lambda expressions

- How does

```
push        ebp
mov         ebp,esp
sub         esp,44h
mov         dword ptr [this],ecx
mov         eax,292B20h
mov         esp,ebp
pop         ebp
ret
```

Assemb

```
    bool
push
push
lea
call
mov
    Sort(n
lea         ecx,[
call        <lambda_d87c98bc2…>::operator bool (__cdecl*)(int,int) (0293270h)
push        eax
push        6
lea         eax,[numbers]
push        eax
call        Sort (02912F3h)
add         esp,0Ch
    return 0;
xor         eax,eax
```

Lambda expressions

- How does

Assemb

```
    bool
push
push
lea
call
mov
    Sort(n
lea          ecx,[
call         <lambda_d87c98bc2…>::operator bool (__cdecl*)(int,int) (0293270h)
push         eax
push         6
lea          eax,[numbers]
push         eax
call         Sort (02912F3h)
add          esp,0Ch
    return 0;
xor          eax,eax
```

```
push         eb
mov          eb
sub          esp,
mov          dword ptr [this],ecx
mov          eax,292B20h
mov          esp,ebp
pop          ebp
ret
```

Address of the actual function that can compare two integers.

Lambda expressions

- This means that this code translates as follows:

App.cpp	Translated code

<pre>int main()
{
 int numbers[] = { 1, 4, 2, 6, 1, 3 };
 auto f = [=] (int n1, int n2) { return n1>n2; };
 bool res = f(1, 2);
 Sort(numbers, sizeof(numbers) / sizeof(int), f);
 return 0;
}</pre>
	<pre>bool lambda_d87c98bc2_function(int n1,int n2) {
	 return n1>n2;
	}
	
	class lambda_d87c98bc2 {
	public:
	 bool operator() (int n1,int n2) {
	  return n1>n2;
	 }
	 operator bool (*)(int,int) {
	  return lambda_d87c98bc2_function;
	 }
	}
	
	int main()
	{
	 int numbers[] = { 1, 4, 2, 6, 1, 3 };
	 lambda_d87c98bc2 f;
	 bool res = f(1, 2);
	 Sort(numbers, sizeof(numbers) / sizeof(int), f);
	 return 0;
	}</pre>

Lambda expressions

- This means that this code translates as follows:

App.cpp	Translated code

App.cpp

```
int main()
{
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	auto f = [=] (int n1, int n2) { return n1>n2; };
	bool res = f(1, 2);
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

Translated code

```
bool lambda_d87c98bc2_function (int n1,int n2) {
	return n1>n2;
}

class lambda_d87c98bc2 {
public:
	bool operator() (int n1,int n2) {
		return n1>n2;
	}
	operator bool (*)(int,int) {
		return lambda_d87c98bc2_function;
	}
}

int main()
{
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	lambda_d87c98bc2  f;
	bool res = f(1, 2);
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

```
auto ptrFunction = (bool (*)(int,int)) f;
Sort (numbers, sizeof(numbers), ptrFunction)
```

Lambda expressions

- This means that this code translates as follows:

App.cpp

```
int main()
{
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	auto f = [=] (int n1, int n2) { return n1>n2; };
	bool res = f(1, 2);
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

Translated code

```
bool lambda_d87c98bc2_function (int n1,int n2) {
	return n1>n2;
}

class lambda_d87c98bc2 {
public:
	bool operator() (int n1,int n2) {
		return n1>n2;
	}
	operator bool (*)(int,int) {
		return lambda_d87c98bc2_function;
	}
}

int main()
{
	int numbers[] = { 1, 4, 2, 6, 1, 3 };
	lambda_d87c98bc2  f;
	bool res = f(1, 2);
	Sort(numbers, sizeof(numbers) / sizeof(int), f);
	return 0;
}
```

```
auto ptrFunction = (bool (*)(int,int)) f;
Sort (numbers, sizeof(numbers), ptrFunction)
```

- Lambdas and STL

Lambda expressions

- A lambda expression can be used with STL (algorithm templates). The following code prints all elements from the vector "v"

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
      vector<int> v = { 1, 2, 3, 5, 6, 7 };
      std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

- The following code doubles the value of all values from vector "v"

App.cpp

```
#include <vector>
#include <algorithm>
int main()
{
      vector<int> v = { 1, 2, 3, 5, 6, 7 };
      std::for_each (v.begin(), v.end(), [](int &value){ value *= 2; });
      std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

Lambda expressions

- A lambda expression can be used with STL (algorithm templates). The following code prints all elements from the vector “v”

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
      vector<int> v = { 1, 2, 3, 5, 6, 7 };
      std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

- The following code doubles the value of all values from vector “v”

App.cpp

```
#include <vector>
#include <algorithm>
int main()
{
      vector<int> v = { 1, 2, 3, 5, 6, 7 };
      std::for_each (v.begin(), v.end(), [](int &value){ value *= 2; });
      std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

This app will print: 2,4,6,10,12,14

Lambda expressions

- Compute the number of odd numbers from a list:

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
	vector<int> v = { 1, 2, 3, 5, 6, 7 };
	int odd_numbers = std::count_if (v.begin(), v.end(), [](int value) { return value % 2 == 0; });
	printf("%d\n", odd_numbers); // 2
}
```

- The following code removes all odd numbers from a list:

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
	vector<int> v = { 1, 2, 3, 5, 6, 7 };
	v.erase(
		std::remove_if (v.begin(), v.end(), [](int value) { return value % 2 == 0; }),
		v.end()
	);
	// v = 1,3,5,7
}
```

Lambda expressions

- A lambda expression can be used with STL (***std::function***) component to describe a function

App.cpp

```
#include <functional>
int main(){
      std::function<int(int, int)> fnc = [](int a, int b)->int { return a + b; };
      printf("%d\n", fnc(10, 20));
}
```

- Usually this type of code is meant to replace a pointer to a function:

App.cpp

```
#include <functional>
typedef int (TypeIntegerSum) (int ,int );

int main()
{
      std::function<TypeIntegerSum> fnc = [](int a, int b)->int { return a + b; };
      printf("%d\n", fnc(10, 20));
}
```

Lambda expressions

- Instead of using pointers to a function, one can replace them with:

App.cpp

```
#include <functional>
typedef int (TypeCompare) (int ,int );

void Sort(int *numbers, int count, std::function<TypeCompare> &compareFunction)
{
}

int main(){
	int n[] = { 1, 2, 3, 4 };
	std::function<TypeCompare> cmpFunction;
	cmpFunction = [](int a, int b)->int { return a > b ? 1 : (a < b?(-1):0); };
	Sort(n, 4, cmpFunction);
}
```

- In this case we replace the standard **int (*)(int,int)** function with std::function

Lambda expressions

- Instead of using pointers to a function, one can replace them with:

App.cpp

```
#include <functional>
typedef int (TypeCompare) (int ,int );
void Sort(int *numbers, int count, std::function<TypeCompare> &compareFunction) { … }
int main(){
	int x = -1;
	int y = 1;
	int n[] = { 1, 2, 3, 4 };
	std::function<TypeCompare> cmpFunction;
	cmpFunction = [x, y] (int a, int b)->int { return a > b ? x : (a < b ? y:0); };
	Sort(n, 4, cmpFunction);
}
```

- The main advantage in this case is that you can pass lambdas that have a caption (while in case of a cast to a function pointer <u>you can not</u>).
- The disadvantage is that using **std::function** is slower than using the pointer to a function directly.

Lambda expressions

- A lambda function and ***std::function*** can also be used with classes:

App.cpp

```
#include <functional>
class Student {
private:
	const char *Name;
	int Grade;
public:
	Student(const char *n, int g) { Name = n; Grade = g; }
	std::function<void()> GetIncrementFunction()
	{
		auto la = [&]() { Grade++; };
		std::function<void()> fnc = la;
		return fnc;
	}
};
int main() {
	Student s("Popescu", 8);
	auto fnc = s.GetIncrementFunction();
	fnc();
	return 0;
}
```

- After the execution of this code, ***s.Grade*** will be 9

Lambda expressions

- Be careful when using ***std::function*** with lambdas that capture local variables *by reference* !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction()
{
	int a = 100;
	printf("Address of a = %p\n");
	std::function<void(int)> fnc = [&a](int value) {
		printf("Address of a (from lambda) = %p\n“, &a);
		a += value;
	};
	return fnc;
}
int main(){
	auto f = GetFunction();
	f(10);
	return 0;
}
```

Lambda expressions

- Be careful when using ***std::function*** with lambdas that capture local variables *by reference* !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction()
{
	int a = 100;
	printf("Address of a = %p\n");
	std::function<void(int)> fnc = [&a](int value) {
		printf("Address of a (from lambda) = %p\n", &a);
		a += value;
	};
	return fnc;
}
int main(){
	auto f = GetFunction();
	f(10);
	return 0;
}
```

```
Address of a = 00DE12D5
Address of a (from lambda) = 00DE12D5
```

- In this case a value on the stack (from the ***GetFunction*** stack) is modified outside ***GetFunction*** !

Lambda expressions

- Be careful when using std::function with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
      int a = 100;
      std::function<void(int)> fnc = [&a](int value) { a += value; };
      return fnc;
}
void Test(std::function<void(int)> &fnc){
      int b = 10;
      std::function<void(int)> temp_fnc = [&b](int value) {  };
      fnc(5);
      printf("b = %d", b);
}
int main(){
      auto f = GetFunction();
      Test(f);
      return 0;
}
```

```
/permissive- /GS- /analyze- /W3 /Zc:wchar_t /ZI /Gm- /Od /sdl
/Fd"Debug\vc141.pdb" /Zc:inline /fp:precise /D "WIN32" /D "_DEBUG"
/D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt
/WX- /Zc:forScope /RTCu /arch:IA32 /Gd /Oy- /MDd /FC /Fa"Debug\"
/nologo /Fo"Debug\" /Fp"Debug\TestCpp.pch" /diagnostics:classic
```

- What will be printed on the screen upon the execution of this code ?
- Tested with cl.exe (19.16.27030.1), VS 2017

Lambda expressions

- Be careful when using std::function with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
      int a = 100;
      std::function<void(int)> fnc = [&a](int value) { a += value; };
      return fnc;
}
void Test(std::function<void(int)> &fnc){
      int b = 10;
      std::function<void(int)> temp_fnc = [&b](int value) {  };
      fnc(5);
      printf("b = %d", b);
}
int main(){
      auto f = GetFunction();
      Test(f);
      return 0;
}
```

- The code compiles and prints **15** on the screen (even if b is 10). Similar results are highly dependent on the stack alignment.

Lambda expressions

- Be careful when using **std::function** with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
	int a = 100;
	std::function<void(int)> fnc = [&a](int value) { a += value; };
	return fnc;
}
void Test(std::function<void(int)> &fnc){
	int b = 10;
	std::function<void(int)> temp_fnc = [&b](int value) {  };
	fnc(5);
	printf("b = %d", b);
}
int main(){
	auto f = GetFunction();
	Test(f);
	return 0;
}
```

The same stack layout !

- The code compiles and prints **15** on the screen (even if b is 10). Similar results are highly dependent on the stack alignment.

Lambda expressions

- Be careful when using **std::function** with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
	int a = 100;
	std::function<void(int)> fnc = [&a](int value)
	return fnc;
}
void Test(std::function<void(int)> &fnc){
	int b = 10;
	std::function<void(int)> temp_fnc = [&b](int value) {  };
	fnc(5);
	printf("b = %d", b);
}
int main(){
	auto f = GetFunction();
	Test(f);
	return 0;
}
```

Notice that ***temp_fnc*** is not used at all !

- The code compiles and prints **15** on the screen (even if b is 10). Similar results are highly dependent on the stack alignment.

Lambda expressions

- Be careful when us… …riables by reference !

App.cpp

```
#include <functional>
std::function<void(int
      int a = 100;
      std::function<voi
      return fnc;
}
void Test(std::function<v            &fnc){
      int b = 10;
      std::functio    oid(int)> temp_fnc = [&b](int value) {  };
      fnc(5);
      printf("b = %d", b);
}
int main(){
      auto f = GetFunction();
      Test(f);
      return 0;
}
```

As local variable "b" from Test function and local variable "a" from GetFunction function have the same location in the stack, calling the lambda function from fnc will affect both of them.

- The code compiles and prints **15** on the screen (even if b is 10). Similar results are highly dependent on the stack alignment.

Using lambda with templates

- (Generic lambdas)

Lambda expressions

- Starting from C++14, lambda expressions can be used with auto parameters creating a template lambda.

App.cpp

```
int main()
{
	auto f = [](auto x, auto y) { return x + y; };
	printf("%d\n",f(10, 20));
	printf("%lf\n",f(10.5, 20.7));
	return 0;
}
```

- The code compiles and prints: 30 and 31.2 into the screen. It only works for the standard C++14 and above.
- In this case the auto parameters work as a template (**this is not however a template !**)

Lambda expressions

- Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
    auto f = []<typename T> (T v1, T v2) { return v1 + v2; };
    printf ("%d %lf\n", f(10, 20), f(1.2,4.3));
    return 0;
}
```

- The code is no different than using auto, the main difference being that we can force a specific type , or a template of a specific type to the lambda expression.
- This code works with g++, but will not compile for cl.exe (VS 2017)
- The code will print 30 and 5.5

Lambda expressions

- Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
    auto f = []<typename T> (T v1, T v2) { return v1 + v2; };
    printf ("%d \n", f(10, 20.5));
    return 0;
}
```

- The code will not compile. The compiler fails to deduce type T (it can either be *int* or *double*) for the call f(10, 20.5)

Lambda expressions

- Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
 auto f = []<typename T> (T v1, T v2) {
        T temp;
        temp = v1 + v2;
        return (int)temp;
    };
    printf("%d\n", f(1.5, 2.2));
    return 0;
}
```

- In this case, we can force the lambda expression to return an *int* value (regardless of the type *T*).
- The result will be 3 (3.7 converted to int).

Lambda expressions

- Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
 auto f = []<typename T, typename R> (T v1, T v2) -> R {
        T temp;
        temp = v1 + v2;
        return (int)temp;
    };
    printf("%d\n", f(1.5, 2.2));
    return 0;
}
```

- In this case, the code will not compile, as type *R* can not be deduced. The problem is located in the fact that we specify that the result type is *R* but the compiler can not deduce it !
- Currently, using an explicit template for f (ex: ***f<double,int>)*** is not supported !

- Mutable capture

Lambda expressions

- Let’s assume the following code:

App.cpp

```
int main()
{
	int a = 0;
	auto f = [&a](int x, int y) { a = x + y; };
	f(10, 20);
	printf("a = %d\n", a);
	return 0;
}
```

- The code compiles and prints “a = 30” on the screen.
- The capture “a” is done via a reference and it can be modified.

Lambda expressions

- Let's assume the following code:

App.cpp

```
int main()
{
	int a = 0;
	auto f = [a](int x, int y) { a = x + y; };
	f(10, 20);
	printf("a = %d\n", a);
	return 0;
}
```

error C3491: 'a': a by copy capture cannot be modified in a non-mutable lambda

- The code however, will NOT work.
- But, as "a" is copied in the capture of the lambda expression, it should work like a class member and therefor we should be able to modify it (even if this will NOT affect the local variable "a" from the main function).
- What happens ?

Lambda expressions

- Let's look at the translated code:

App.cpp

```
int main() {
      int a = 0;
      auto f = [a](int x, int y) { a = x + y; };
      f(10, 20);
      printf("a = %d\n", a);
}
```

Translated code

```
class lambda_3c006326 {
      int a;
public:
      lambda_3c006326(int &ref_a): a(ref_a) {  }
      void operator() (int x,int y) const { a = x + y; }
};
int main() {
      int a, b;
      lambda_3c006326 f (a);
      f(10, 20);
      printf("a = %d\n", a);
}
```

Lambda expressions

- Let’s look at the translated code:

App.cpp

```
int main() {
      int a = 0;
      auto f = [a](int x, int y) { a = x + y; };
      f(10, 20);
      printf("a = %d\n", a);
}
```

Translated code

```
class lambda_3c006326 {
      int a;
public:
      lambda_3c006326(int &ref_a): a(ref_a) { }
      void operator() (int x,int y) const { a = x + y; }
}
int main() {
      int a, b;
      lambda_3c006326 f (a);
      f(10, 20);
      printf("a = %d\n", a);
}
```

The problem lies in the way operator() is defined !

Because of the “*const*” operator from the end of the definition, operator() can not modify any of its data members.

Lambda expressions

- The solution is to use "**mutable**" keyword when defining the lambda:

App.cpp

```
int main() {
      int a = 0;
      auto f = [a](int x, int y) mutable { a = x + y; };
      f(10, 20);
      printf("a = %d\n", a);
}
```

Translated code

```
class lambda_3c006326 {
      mutable int a;
public:
      lambda_3c006326(int &ref_a): a(ref_a) {  }
      void operator() (int x,int y) const { a = x + y; }
}
int main() {
      int a, b;
      lambda_3c006326 f (a);
      f(10, 20);
      printf("a = %d\n", a);
}
```

As a result, the created class has the internal data members defined as mutable (meaning that even if operator() is const, those data members can still be modified).

Lambda expressions

- The solution is to use “**mutable**” keyword when defining the lambda:

App.cpp

```
int main() {
      int a = 0;
      auto f = [a](int x, int y) mutable { a = x + y; };
      f(10, 20);
      printf("a = %d\n", a);
      return 0;
}
```

- The cod compiles and prints “a = 0” into the screen.
- Even if the “**mutable**” keyword is used, as “**a**” was captured by making a copy of it and only the copy is modified when calling **f(10,20)**

Lambda expressions

- The solution is to use “**mutable**” keyword when defining the lambda:

App.cpp

```
int main()
{
	int index = 0;
	auto counter = [index] () mutable { return index++; };
	for (int tr = 0; tr < 10; tr++)
	{
		printf("%d,", counter());
	}
	return 1;
}
```

- The code compiles and prints 0,1,2,3,4,5,6,7,8,9 into the screen.

Lambda expressions

- The solution is to use “**mutable**” keyword when defining the lambda:

App.cpp

```
int main()
{
	int index = 0;
	auto counter = [index] () mutable -> int { return index++; };
	for (int tr = 0; tr < 10; tr++)
	{
		printf("%d,", counter());
	}
	return 1;
}
```

- If we want to describe the type of la lambda, then the **mutable** keyword should be added before the type (after the lambdas parameters)

Initialized lambda capture

Lambda expressions

- C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
	int a = 10, b = 20;

	auto f1 = [var1 = a+b, var2 = a-b] (int x, int y) { return x + y + var1 + var2; };

	printf("%d\n", f1(1, 2));
	return 0;
}
```

- In this case lambda "f1" has two variable captured (var1 and var2). "var1" equals 10+20 = 30, and "var2" equals 10-20 = -10;
- The code compiles under C++14 standards and prints 1+2+30-10 = 23

Lambda expressions

- C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
	auto f1 = [counter = 0] () mutable { return counter++; };
	for (int tr=0;tr<10;tr++)
	{
		printf("%d\n",f1());
	}
	return 0;
}
```

- This type of initialization can be used to create lambdas with their own parameters. In this example, "f1" has one member (counter) that is initialized with 0 and incremented each time f1() is called.
- The code prints the numbers from 0 to 9
- ***mutable*** specifier is a MUST for this initialization to work. Without it, the code will produce a compile error as ***counter*** can not be modified.

Lambda expressions

- C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
	auto f1 = [counter = 1]() mutable { counter*=2; return counter; };
	for (int tr=0;tr<10;tr++)
	{
		printf("%d\n",f1());
	}
	return 0;
}
```

- In this case the type of counter is deduced to be int.
- The code compiles and prints 2,4,8,16,32,64,128,256,512,1024

Lambda expressions

- C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
	auto f1 = [unsigned char counter = 1]  () mutable { counter*=2; return counter; };
	for (int tr=0;tr<10;tr++)
	{
		printf("%d\n",f1());
	}
	return 0;
}
```

- This code will not work → it is not allowed to set the type of capture. Type of capture is deduced from the assignment.

Lambda expressions

- C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
	auto f1 = [counter = (unsigned char) 1]() mutable { counter*=2; return counter; };
	for (int tr=0;tr<10;tr++)
	{
		printf("%d\n",f1());
	}
	return 0;
}
```

- However, you can force the type of such assignments by forcing the type of the evaluated value (usually using a cast).
- The code will compile and will print: 2,4,8,16,32,64,128,0,0,0
- The last 3 zeros are because counter is of type **unsigned char** and once it reaches value 256 it overflows and becomes 0.

Lambda expressions

- C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
    auto random = [seed = 1U] (unsigned int maxNumber) mutable {
        seed = 22695477U * seed + 1;
        return seed % maxNumber;
    };

    for (int tr=0;tr<10;tr++)
        printf("Pseudo random number between 0 and 99: %d \n", random(100));
    return 0;
}
```

- This example generates a pseudo random function based on the Linear congruential generator algorithm.
- The function uses the **seed** internal variable to generate the next random number.

New feature in C++17 and beyond

Lambda expressions

- C++17 standards allows to capture (***this**) → by using its copy constructor

App.cpp

```
struct MyClass
{
    int a;
    MyClass(int value) { a = value; }
    MyClass(const MyClass & obj) { std::cout << "Copy ctor" << std::endl; a = obj.a; }
    auto GetLambda() { return [this]() { std::cout << a << std::endl; }; }
};
int main()
{
    MyClass c = 10;
    auto f = c.GetLambda();
    f();
    c.a += 10;
    f();
    return 0;
}
```

- This code creates a lambda functions that captures ***this*** (as a pointer). The execution will print 10 and 20 on the screen (as lambda captures a reference to object *c*).

Lambda expressions

- C++17 standards allows to capture (***this**) → by using its copy constructor

App.cpp

```
struct MyClass
{
    int a;
    MyClass(int value) { a = value; }
    MyClass(const MyClass & obj) { std::cout << "Copy ctor" << std::endl; a = obj.a; }
    auto GetLambda() { return [ *this ]() { std::cout << a << std::endl; }; }
};
int main()
{
    MyClass c = 10;
    auto f = c.GetLambda();
    f();
    c.a += 10;
    f();
    return 0;
}
```

- This code only works on C++17 standard. In this case, lambda captures a copy of c object, and the results printed on the screen will be: "**Copy ctor**", then "**10**" and then "**10**" again (only the local *c* object is modified, not its copy).

Lambda expressions

- C++17 standards also allows creating a **constexpr** lambda expression

App.cpp

```
int main()
{
    int a = 10;
    constexpr auto f = [](int v1, int v2) constexpr { return v1 + v2; };
    printf("%d\n", f(1, 2));
    return 0;
}
```

- This code only works on C++17 standard. However, the generated cod does not show that the **constexpr** optimization is indeed applied. Code was tested with cl.exe, version 19.16.27025.1 and 19.16.27030.1 for x86 architecture

Generated assembly code for "*printf("%d\n", f(1,2)*"

```
push   2
push   1
lea    ecx,[f]
call   <lambda_f97ecf0fb43f36dc04c4d496e5fe74ab>::operator()
push   eax
push   offset string "%d\n"
call   printf
add    esp,8
```

Lambda expressions

- C++17 standards also allows creating a **constexpr** lambda expression

App.cpp

```
int main() {
    int a = 10;
    constexpr auto f = [](int v1, int v2) constexpr { return v1 + v2; };
    int aa[f(1, 2)];
    aa[0] = 0;
    return 0;
}
```

- It however works for the previous case, and “aa” local variable is instantiated.
- It also works with **static_assert** like in the next example.

App.cpp

```
int main()
{
    constexpr auto f = [](int v1, int v2) constexpr { return v1 + v2; };
    static_assert(f(1, 2) == 3);
    return 0;
}
```

Q & A