NEW YORK UNIVERSITY

ABU DHABI

1. Scope & Lifetime

2. Namespace

NEW YORK UNIVERSITY

ABU DHABI

1. Scope & Lifetime

2. Namespace

Scope & Lifetime

NEW YORK UNIVERSITY
ABU DHABI

- The scope of a declaration is the part of the program for which the declaration is in effect.
  - Global Scope
  - Local Scope
- The lifetime of a variable or object is the time period in which the variable/object has valid memory.

Scope of variables

NEW YORK UNIVERSITY
ABU DHABI

- A local variable can be used only in the function where it is defined
  - i.e., the scope of a local variable is the current function.
- A global variable can be used in all functions below its definition
  - i.e., the scope of a global variable is the range from the variable definition to the end of the file.

Global vs. local Variable

NEW YORK UNIVERSITY
ABU DHABI

```
#include <stdio.h>
#define PI 3.14

int count;

int func()
{
   int i, j;

   scanf("%d %d", &i, &j);
   count += i+j;
   return 0;
}

int main()
{
   int i;

   func();
   for (i=0; i<count; i++)
      printf("%d",i);
   return 0;
}
```

Global Variable

Local Variables of func()

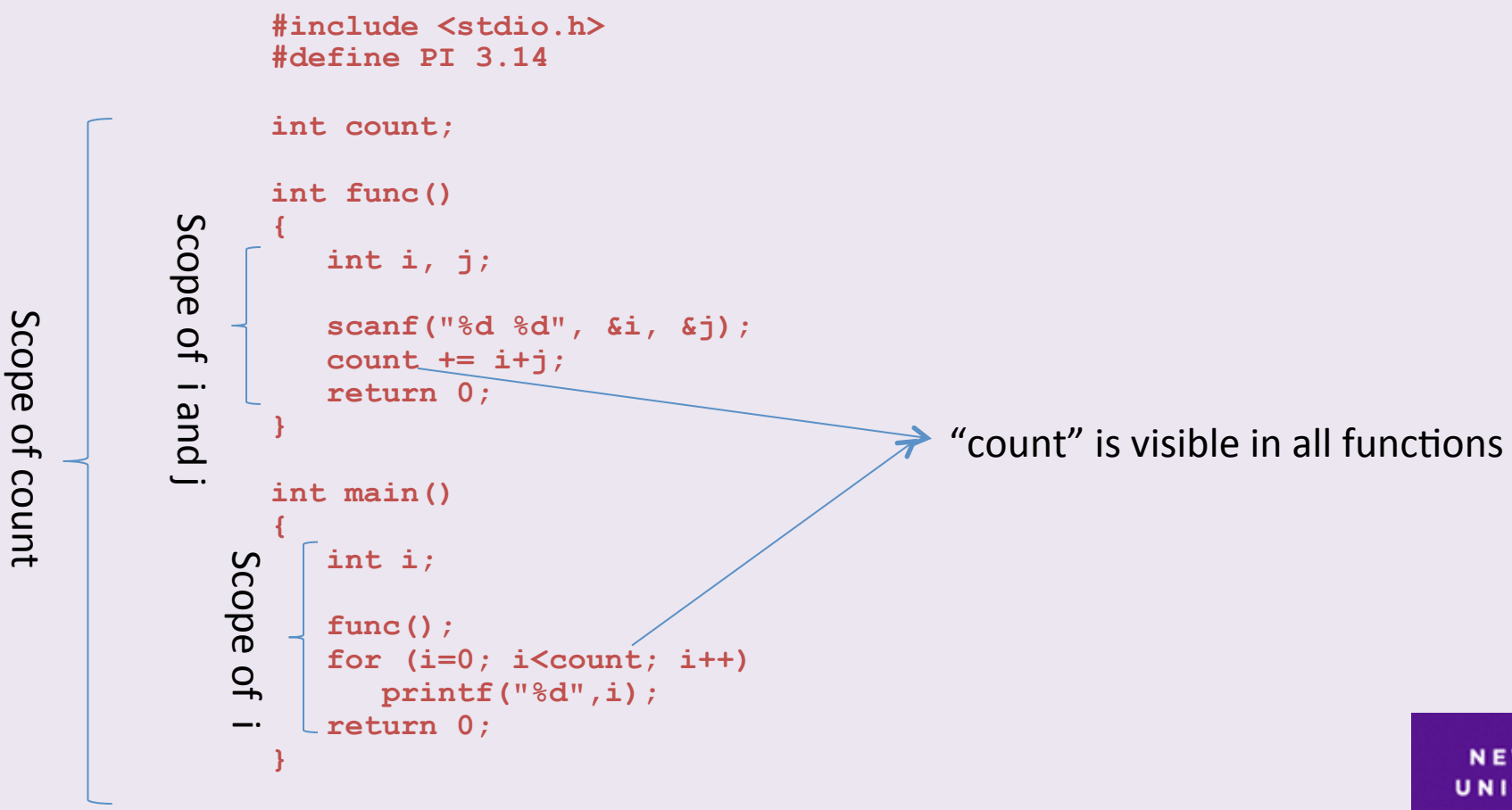
Local Variable of main()

Lifetime of variables

NEW YORK UNIVERSITY
ABU DHABI

- The lifetime of a variable depends on its scope.
- A local variable is alive as long as the function where it is defined is active.
  - When the function terminates, all of the local variables (and their values) are lost.
  - When the function is called again, all variables start from scratch; they don't continue with their values from the previous call (except for local static variables which are not discussed).
- The lifetime of a global variable is equivalent to the lifetime of the program.

Scope and lifetime of variables



NEW YORK UNIVERSITY

ABU DHABI

Summary

NEW YORK UNIVERSITY
ABU DHABI

Global variables	Local variables
Visible in all functions	Visible only within the function they are defined
Zero by default	Uninitialized
A change made by a function is visible everywhere in the program	Any changes made on the value are lost when the function terminates (since the variable is also removed)
Scope extends till the end of the file	Scope covers only the function
Lifetime spans the lifetime of the program	Lifetime ends with the function

Changing local variables

NEW YORK UNIVERSITY
ABU DHABI

- Any change made on local variables is lost after the function terminates.

```
void f()
{   int a=10;
    a++;
    printf("in f(): a=%d\n",a);
}
int main()
{   int a=5;
    f();
    printf("After first call to f(): a=%d\n",a);
    f();
    printf("After second call to f(): a=%d\n",a);
}
```

Changing global variables

NEW YORK UNIVERSITY ABU DHABI

- Any change made on global variables remains after the function terminates.

```
int b;
void f()
{
   b++;
   printf("in f(): b=%d\n",b);
}
int main()
{
   f();
   printf("After first call to f(): b=%d\n",b);
   f();
   printf("After second call to f(): b=%d\n",b);
}
```

This is called side effect. Avoid side effects.

Changing value parameters

NEW YORK UNIVERSITY
ABU DHABI

- Any change made on value parameters is lost after the function terminates.

```
void f(int c)
{
    c++;
    printf("in f(): c=%d\n",c);
}
int main()
{   int c=5;
    f(c);
    printf("After f(): a=%d\n",c);
}
```

Local variable definition veils global definition

- A local variable with the same name as the global variable veils the global definition.

```
int d=10;
void f()
{
    d++;
    printf("in f(): d=%d\n",d);
}
int main()
{   int d=30;
    f();
    printf("After first call to f(): d=%d\n",d);
    f();
    printf("After second call to f(): d=%d\n",d);
}
```

NEW YORK UNIVERSITY

ABU DHABI

Parameter definition veils global definition

- Similar to local variables, parameter definition with the same name as the global variable also veils the global definition.

```
int e=10;
void f(int e)
{
   e++;
   printf("in f(): d=%d\n",e);
}
int main()
{  int g=30;
   f(g);
   printf("After first call to f(): g=%d\n",g);
}
```

NEW YORK UNIVERSITY

ABU DHABI

NEW YORK UNIVERSITY

ABU DHABI

1. Scope & Lifetime

2. Namespace

Namespace

NEW YORK UNIVERSITY
ABU DHABI

Namespace: A separate region for a group of variables, functions and classes and so on and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

Why Namespace?

NEW YORK UNIVERSITY
ABU DHABI

1. You define a new function named foo() and there is another existing library which already defines function with the same name foo().

2. In this case, the compiler does not know which version of foo() function you are referring to within your code.

Definition of Namespace?

A namespace definition begins with the keyword **namespace** followed by the namespace name

```
namespace namespace_name
{
   // code declarations
}
```

To call the a function or refer to a variable in a namespace, prepend the namespace name :

```
name::reference;
// reference could be variable or function.
```

Using Namespace?

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

// first name space
namespace space_A{
   void foo(){
      cout << "Inside space_A" << endl;
   }
}
// second name space
namespace space_B{
   void foo(){
      cout << "Inside space_B" << endl;
   }
}
int main ()
{

   // Calls function from space_A.
   space_A::foo();

   // Calls function from space_B.
   space_B::foo();

   return 0;
}
```

```
Inside space_A
Inside space_B
```

Directive using namespace

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

// first name space
namespace space_A{
   void foo(){
      cout << "Inside space_A" << endl;
   }
}
// second name space
namespace space_B{
   void foo(){
      cout << "Inside space_B" << endl;
   }
}

using namespace space_A;
int main ()
{

   // Calls function from space_A.
   foo();
   return 0;
}
```

```
Inside space_A
```

Nested Namespaces

```
namespace namespace_name1
{
    // code declarations
    namespace namespace_name2
    {
        // code declarations
    }
}
```

Access to sub-namespace

```
// to access sub-namespace
using namespace namespace_name1::namespace_name2;
```

A function Example

Namespace

Function Template

Inline keyword

Function overloading

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

// Function tempalte
template <typename T>
inline T square(T x)
{
   T result;
   result = x * x;
   return result;
};

// Function Overloading by different types of parameters
int multiply(int x, int y)
{
  return x * y;
};

float multiply(float x, float y)
{
  return x * y;
};

// Function Overloading by different number of parameters
float multiply(float x, float y, float z)
{
  return x * y * z;
};
```

```
int main()
{
   int    i, ii;
   float  x, xx;
   float y, yy;

   i = 2;
   x = 2.2;
   y = 2.2;

   //Overloading function
   cout << multiply(i,i) << endl;
   cout << multiply(x,y) << endl;
   cout << multiply(x,y,z) << endl;

   // Explicit use of template
   ii = square<int>(i);
   cout << i << ": " << ii << endl;

   xx = square<float>(x);
   cout << x << ": " << xx << endl;

   // Implicit use of template
   yy = square(y);
   cout << y << ": " << yy << endl;

   return 0;
}
```