IDK1531 Advanced C++ Course

Types

Aleksandr Lenin

Tallinn University of Technology

February 11th, 2019

Fundamental Types

void is an **incomplete** type with an empty set of values. This type cannot be completed, objects of type void, arrays of elements of type void and references to type void are disallowed.

std::nullptr_t is the type of the **null pointer** literal nullptr.

bool is a type capable of storing true and false values.

Integral Types

int has width of at least 16 bits. In 32/64 bit systems is it common for int to occupy at least 32 bits.

Size modifiers:

short – type will be optimized for space and will have width at least 16 bits.

long – type will have at least 32 bits.

long long – type will have at least 64 bits.

If any size modifiers are used, the int keyword may be omitted.

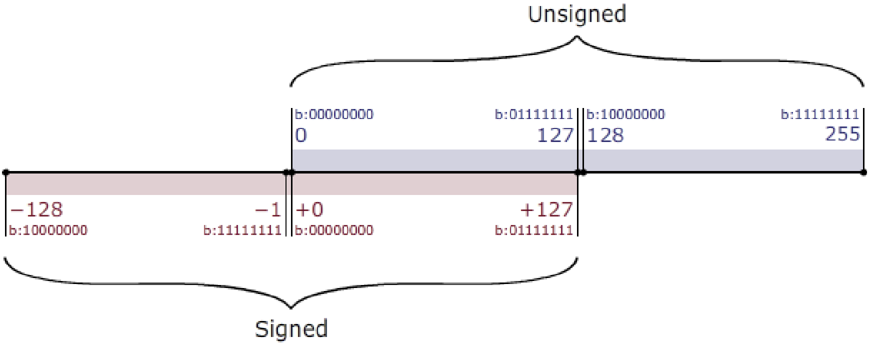
Common data models are the following:

- **ILP32** or **4/4/4**. int, long and pointer size is 32 bit.
  - 32 bit OS (Microsoft Windows, Unix and Unix-like systems
  - Win32 API
- **LLP64** or **4/4/8**. int and long size is 32 bits, pointer size is 64 bits.
  - 64 bit Microsoft Windows
  - Win64 API
- **LP64** or **4/8/8**. int size is 32 bits, long and pointer size is 64 bits.
  - 64 bit Unix and Unix-like systems (Linux, Mac, *BSD, ...)

Two sigedness modifiers:

signed – type for sign representation, the most significant bit is reserved to represent the sign.

unsigned – type for unsigned representation.



Computations using **unsigned** integral types are performed **modulo the size of the value space**.

I.e., adding two unsigned int type variables a and b is computed as $a + b \bmod 2^{32}$.

Adding two unsigned char type variables a and b is computed as $a + b \bmod 2^{8}$.

Example

```
unsigned char a = 100;
unsigned char b = 200;
unsigned char c = a + b;
std::cout << (int) c << std::endl;          // prints 44
std::cout << (300 % 256) << std::endl;  // prints 44
```

Overflowing a **signed** type results in **undefined behavior**.

Example

```
char d{127}; d++;
std::cout << (int) d << std::endl;   // prints −128. This is UB.
```

Operations between signed and unsigned integers produce an unsigned result.

Example

```
unsigned a = 10;
int  b = −15;
std::cout << (a+b) << std::endl;   // prints 4294967291
```

Fixed width integer types are the following:

int8_t, int16_t, int32_t, int64_t – signed integer types with width exactly $8, 16, 32, 64$ bits.

uint8_t, uint16_t, uint32_t, uint64_t – unsigned integer types with width exactly $8, 16, 32, 64$ bits.

[u]int_fast8_t, [u]int_fast16_t, [u]int_fast32_t, [u]int_fast64_t – fastest signed integer type with width of **at least** $8, 16, 32, 64$ bits.

[u]int_least8_t, [u]int_least16_t, [u]int_least32_t, [u]int_least64_t – smallest integer type with width $8, 16, 32, 64$ bits.

Character Types

char – the type for character representation that can be efficiently processed by the target system.

The sigedness of char depends on the compiler and the target platform. The char type defaults to

- unsigned char on ARM, PowerPC architectures
- signed char on Intel x86 and x86_64 architectures.

wchar_t – a type for wide character representation.

Usually has size 32 bits, sufficient to represent the entire Unicode character set.

Exception: Windows. The size of wchar_t is 16 bits, and it can encode UTF-16 character set.

Fixed size character types are the following.

char8_t – type for UTF-8 character set representation.

char16_t – type for UTF-16 character set representation.

char32_t – type for UTF-32 character set representation.

The C++ standard guarantees that:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long).
```

Floating Point Types

Depending on FPU coprocessor, typically these types are:

float – IEEE-754 32 bit single precision floating point type.

double – IEEE-754 64 bit double precision floating point type.

long double – extended precision floating point type.

The type long double is not regulated by IEEE-754 and depends on the compiler and target architecture. On Intel x86 and x86_64 architectures, the type long double defaults to 80 bit x87 floating point type.

Floating point numbers support special values

- infinity to represent a positive infinity.
- nan to represent a NaN.

CV Type Qualifiers

For any type T, including incomplete types, excluding reference and function types, there are 3 possible specifications of type T, namely:

1. const T – const-qualified type T. An object of a constant type cannot be modified. An attempt to modify such an object directly results in a compile-time error, attempt to modify it indirectly (through a reference or a pointer) results in undefined behavior.
2. volatile T – volatile-qualified type T. Prevents the compiler from optimizing the code where such a type is present, since it is assumed that such a type may be changed and the compiler may not be aware of it.
3. const volatile T – const-volatile-qualified type T. An object behaves as a constant and volatile object.

Type conversions w.r.t. cv-qualifications.

unqualified < const < const volatile
unqualified < volatile < const volatile

References and pointers to less cv-qualified types may be implicitly converted to references and pointers to more cv-qualified types.

To convert a pointer or a reference of a more qualified type to a pointer or a reference to a less cv-qualified type, const_cast must be used.

The cv-qualification of an array is the same as the cv-qualification of its elements.

The mutable specifier permits modification of a class member even if the containing object is declared const.

Example

```
const struct {
    int a;
    mutable int b;
} x = {0,0};

x.a++;   // compilation-time error
x.b++;   // mutable, modification is permitted
std::cout << x.a << x.b << std::endl;  // prints 01
```

Reference Types

A **reference** is an **alias** to an existing object or function.

A reference needs to be **initialized** to refer to an object. Uninitialized references result in compilation-time errors.

A reference of some type `T` may be initialized with:

1. an **object** of type `T`.
2. a function of type `T`.
3. an object **implicitly convertible** to type `T`.

Once initialized, the referred object cannot be changed, the reference sticks to the object it refers to.

A reference is initialized:

- when a named lvalue or rvalue reference variable is initialized
- in a function call when one or more arguments are reference type
- when a function returns a reference type
- when a non-static reference type member is initialized

A reference may refer to a **complete type**. I.e., there are no references to void.

Reference is **not an object** and therefore references do not necessarily occupy storage (although some compilers may allocate storage).

For the same reason, there are no references to references.

References are not cv-qualified. A "const reference to type T" is an ordinary reference to type const T.

The **lifetime** of a reference begins when its initialization is complete and ends when the storage duration ends (as if it were a scalar object).

The lifetime of the referred-to object may end **before** the lifetime of the reference. If this happens, such a reference is called a **dangling reference**.

Using dangling references is **undefined behavior**.

An **lvalue** reference declarator

```
T& [attr] identifier
```

declares identifier as an **lvalue reference** to type T.

lvalue references are used to **alias** existing objects or functions, optionally with a different cv-qualification.

Example

```
int x = 5;              // a variable of type int
int& rx = x;            // an lvalue reference to int
const int& crx = x;     // an lvalue reference to const int

std::cout << x << " " << rx << " " << crx << std::endl;  // prints 5 5 5
rx += 2;    // it is ok to assign a new value to an lvalue reference
crx += 2;   // error, cannot assign a read-only reference
std::cout << x << " " << rx << " " << crx << std::endl;  // prints 7 7 7
```

An **rvalue** reference declarator

```
T&& [attr] identifier
```

declares identifier as an **rvalue reference** to type T, optionally with different cv-qualification.

rvalue references can be used to **extend the lifetime of temporary objects**. lvalue references to const can extend the lifetime of temporary objects as well, but are not modifiable through them.

Example

```cpp
int x = 5;                    // integer of type int
const int& lrx = x + x;       // an lvalue reference to const int
int&& rrx = x + x;            // an rvalue reference to int

std::cout << x << " " << lrx << " " << rrx << std::endl;     // prints 5 10 10
lrx += 10;  // error, cannot assign a read-only reference
rrx += 10;  // can modify through reference to non-const
std::cout << x << " " << lrx << " " << rrx << std::endl;     // prints 5 10 20
```

If a reference is bound to a temporary or to a subobject, the lifetime of the temporary is extended to match the lifetime of the reference.

Exceptions to this rule:

- a temporary bound to the `return` value of a function returning a reference is destroyed immediately after the function exits and such a function always returns a dangling reference.
- a temporary bound to a reference argument in a function call exists only in the function scope. If the function returns a reference, it becomes a dangling reference.
- a temporary bound to a reference in the initializer used in a `new` expression exists until the end of the full expression containing that `new` expression, the lifetime is not extended to match the lifetime of the initialized object.

A reference may refer to an object that is equal or less cv-qualified.

Example

```
int x = 2;                   // a variable of type int
int & rx = x;                // equal cv-qualification
const int & crx = x;         // more cv-qualified, ok
int & rrx = crx;             // error: less cv-qualified
const int & rrx = crx;       // ok
```

In the last line, the lvalue reference rrx is not bound to crx (there are no references to references, remember?). rrx is bound to the same object to which crx is bound. In this case, it is int x;

Use const_cast<T> to cast more cv-qualified reference to a less cv-qualified reference.

Example

```
int x = 2;
const int & crx = x;                  // a reference to const int
int & rx = crx;                       // error, less cv-qualified
int& rx = const_cast<int&>(crx);      // ok
```

You may declare lvalue references to functions.

Example

```
void f (int a) { std::cout << a << std::endl; }       // a function of type void(int)
int g() { return 2; }                          // a function of type int(void)
void (&rf)(int) = f;    // an lvalue reference to function f()
int (&rg)() = g;        // an lvalue reference to function g()
```

and references to arrays

Example

```
int data[3];
int (&rdata)[3] = data;
```

With the exception of a const qualified lvalue reference, lvalue references cannot be bound to temporaries, while rvalue references can

Example

```
int& ra = 1;            // error, cannot bind lvalue reference to rvalue
int&& rra = 1;          // ok, bound to rvalue
const int& cra = 1;   // ok, bound to read-only lvalue
```

rvalue references cannot bind to lvalues.

Example

```
int n = 2;
int&& rn1 = n;    // error, cannot bind to lvalue
int&& rn2 = static_cast<int&&>(n);  // ok, cast n to an rvalue
float&& rn3 = n;  // ok, bound to an rvalue temporary 2.0
```

It is possible to create situations in which the lifetime of the referred object ends, but the reference remains accessible. Such cases are referred as **dangling references**. Accessing such a reference is **undefined behavior**.

A common example, is returning a reference to an automatic variable.

Example

```
std::string& f() {
    std::string s("Hello, World!");
    return s;  // s is destroyed
}

std::string& sref = f();  // f() returns a dangling reference
std::cout << sref;        // undefined behavior, read attempt from a dangling reference
```

Temporaries' lifetime restrictions. Consider the following structure

```
struct S { int x; const int& lref; int&& rref; };
```

If initialized as `S s{1,2,3};`, the temporary `2` is bound to `s.lref`, temporary `3` is bound to `s.rref`, the lifetimes of the temporaries is extended to match the lifetime of object `s`.

If initialized as a pointer `S* p = new S{1,2,3}`, the temporary `2` is bound to `s.lref`, temporary `3` is bound to `s.rref`, but the lifetime of the references ended at the end of the `new` statement, and `p->lref` and `p->rref` are **dangling references**.

A function returning a reference to a temporary returns a dangling reference.

Example

```
const int& f() { return 1; }
const int& result = f();      // f() returns a dangling reference
```

Forwarding references is a special kind of references that preserve the value category of a function argument, and makes it possible to **forward** it using std::forward, which

- forwards lvalues as lvalues or rvalues
- forwards rvalues as rvalues
- prohibits forwarding lvalues as rvalues.

Two use cases

1. Function parameter of a function template declared as rvalue reference to cv-unqualified type template parameter.
2. auto&&, except when deduced from a brace-initialized list. auto&& is the safest way to refer to elements in ranged-for loops.

Example

```
for (auto&& e: f()) {
    // e is a forwarding reference
}

auto&& a = {1, 2, 3}; // a is not a forwarding reference
```

Pointer Types

```
[attr] T [cv] * [cv] identifier
```

A **pointer** is an object that stores an **address** of another object or a function in memory.

Implications:

- no pointers to references or bitfields exist
- there exist pointers to pointers
- there exist references to pointers

Every pointer is one of:

- a pointer to an object or a function – stores the address of the first byte occupied by the object storage in memory
- a pointer past the end of an object – stores the address of the first byte after the end of storage occupied by the object
- a null pointer nullptr – stores the zero address
- a invalid (dangling) pointer – a pointer that points at a (nonexistent) object whose lifeteime has ended

Attempts to use an invalid pointer or passing an invalid pointer as an argument to a memory deallocation function is **undefined behavior**.

The "address-of" operator & returns the address of a given object in memory and may be used to initialize a pointer.

The dereference operator $*$ may be used to access the pointed-to object.

Example

```
int x = 10, y = 3;
int* p = &x;         // now p points to x
std::cout << *p;     // dereferencing p, printing 10
p = &y;              // now it points to y
*p = 15;             //dereferencing p, assigning new value to y
std::cout << *p;     // printing the value of y, which is 15 now
int** pp = &p;       // a pointer to p, aka a pointer to a pointer pointing at y
std::cout << *pp;    // printing the address of p
std::cout << **pp;   // printing the value of y
```

For convenience, the −> operator allows to access members of an object via a pointer. The call object−>member is a syntactic sugar, and is equivalent to ∗(obj).member.

Example

```
struct S { int x; } s;
struct S *ps = &s;
s.x = 2;
(*ps).x = 2;
ps->x = 2;
```

A reference to a pointer, as any reference, is used to alias an object.

Example

```
int x = 3, y=5;
int* px = &x;      // px is a pointer pointing at x
int*& rpx = px;    // rpx is a reference to px
rpx = &y;          // now px points at y
*rpx = 15;         // now the value of y is 15
std::cout << *rpx; // prints 15
int*&& rval = new int(5);  // rval is an rvalue reference to a pointer to an integer
std::cout << rval; // prints the address allocated by new and stored in rval
std::cout << *rval; // prints 5 (the initialized value)
delete rval;        // deallocating dynamic memory
```

If const keyword appears on the **left** of ∗, such a pointer points at a **constant type**. You can modify the pointer, but cannot modify the pointed-to data.

Example

```
int x;
const int * px = &x;    // a pointer to const int
int const * px2 = &x;   // a pointer to const int
px2 = nullptr;          // ok
*px = 2;                // error, modification of constant object
```

If const keyword appears on the **right** of ∗, such a pointer is a **constant pointer** that points at a fixed address and cannot be modified. The pointed-to object can still be modified.

Example

```
int x;
int * const px = &x;    // a constant pointer to an integer
px = nullptr;           // error, modification of a constant pointer
*px = 2;                // ok, modification of a pointed-to non-const object
```

Finally, if const appears on **both sides** of the ∗, such a pointer is known as a constant pointer to a constant type. Modification of the pointer, as well as the pointed-to object is not permitted.

Example

```
int x;
int const * const px = &x;    // a constant pointer to a const int
px = nullptr;                 // error, modification of a constant pointer
*px = 2;                      // error, modification of a pointed-to const object
```

Due to implicit array-to-pointer conversion, an array variable is implicitly casted to a pointer to its first element.

Example

```
int x[5]{1,2,3,4,5};
int* px = x;       // px points at the first integer in array x
int* px2 = &x[0]; // px2 points at the first integer in array x
int (*px3)[5] = &x; // px3 poins at an array of 5 integers
```

Any pointer can be implicitly casted into a pointer to void. The inverse conversion requires a static_cast call.

Example

```
char c; short s; int i; long l; long long ll;
void *vpc = &c, *vps = &s, *vpi = &i, *vpl = &l, *vpll = &l;
char* pc = static_cast<char*>(vpc);
short* ps = static_cast<short*>(vps);
int* pi = static_cast<int*>(vpi);
long* pl = static_cast<long*>(vpl);
long long* pll = static_cast<long long *>(vpll);
```

Pointers to functions

Example

```
void f() {}
int g(int a) { return a; }
int h(int k) { return 2*k; }

void (*pf)() = &f;   // a pointer to a function f
void (*pf2)() = f;   // another pointer to a function f
void (*pf3)() = nullptr; // a pointer to type void(void) initialized with zero address
int (*pg)(int) = g;  // a pointer to function g
pg(10);              // g(10) is called
pg = h;              // now pg points at function h
pg(10);              // h(10) is called
```

Pointers, with exception for type void*, support increment and decrement operations.

If a scalar k is added to a pointer of type T, then the pointer will point at a new address, which is shifted by k * sizeof(T) compared to the initial address the pointer was pointing at.

Example

```
char* p = reinterpret_cast<char*>(0x100);
std::cout << (void*) p << std::endl;        // 0x100
std::cout << (void*) (p+1) << std::endl;    // 0x101
std::cout << (void*) (p+2) << std::endl;    // 0x102

int* pi = reinterpret_cast<int*>(0x100);
std::cout << pi << std::endl;               // 0x100
std::cout << (pi+1) << std::endl;           // 0x104
std::cout << (pi+2) << std::endl;           // 0x108
```

The random access operator [] allows to access objects at addresses **relative to** the address stored by the pointer. Let p is a pointer to type T. Then p[i] corresponds to the value at address p + i * sizeof(T).

Example

```
char* pc = reinterpret_cast<char*>(0x100);
short* ps = reinterpret_cast<short*>(0x100);
int* pi = reinterpret_cast<int*>(0x100);

std::cout << (void*) pc << " "                // 0x100
          << (void*) &pc[1] << " "            // 0x101
          << (void*) &pc[2] << " "            // 0x102
          << (void*) &pc[3] << std::endl;     // 0x103

std::cout << ps << " "                        // 0x100
          << &ps[1] << " "                    // 0x102
          << &ps[2] << " "                    // 0x104
          << &ps[3] << std::endl;             // 0x106

std::cout << pi << " "                        // 0x100
          << &pi[1] << " "                    // 0x104
          << &pi[2] << " "                    // 0x108
          << &pi[3] << std::endl;             // 0x10C
```

Given two pointers of the same type, the difference between them yields the number of elements of these types that fit into a given range.

Example

```
struct S { int a,b,c,d; };
void* begin = reinterpret_cast<void*>(0x100);
void* end = reinterpret_cast<void*>(0x120);
std::cout << static_cast<char*>(end) - static_cast<char*>(begin);  // 32
std::cout << static_cast<short*>(end) - static_cast<short*>(begin); // 16
std::cout << static_cast<int*>(end) - static_cast<int*>(begin); // 8
std::cout << static_cast<long*>(end) - static_cast<long*>(begin); // 4
std::cout << static_cast<long long*>(end) - static_cast<long long*>(begin) // 4;
std::cout << static_cast<float*>(end) - static_cast<float*>(begin) // 8;
std::cout << static_cast<struct S*>(end) - static_cast<struct S*>(begin) // 2;
```

The only supported operations with pointers are

- Adding a pointer and a scalar (positive, negative, or zero)
- Subtracting two pointers of the same type

It is illegal to subtract pointers of different types, as well as adding two pointers together. Such attempts will produce compilation time errors.

Arrays

An array declaration declares an object of array type.

```
T name [[expr]] [attr];
```

where:

- T is the type of elements in the array. It can be any fundamental type, pointers, classes, enumerations, and other arrays of the same type. There are no arrays with element type void, no arrays of references, or arrays of functions.
- name is any valid identifier.
- [expr] an optional constant expression convertible to std::size_t which evaluates to a value greater than zero (since C++14).
- [attr] optional attributes.

Example

int a[5]; declares a as an array object consisting of 5 continuously allocated objects of type int.

Applying cv-qualifications to array type applies the qualifiers to element type.

Example

`const int a[5];` declares an array of 5 elements of type `const int`.

If an array is allocated **dynamically** (i.e. using `new` expression), its size is allowed to be zero. Accessing allocated memory block of size 0 is undefined behavior.

Example

```
int *a = new int[0];    // accessing a[0] or *a is UB
delete[] a;             // it is still required to deallocate memory
```

An array `arr` of `N` elements may be accessed using the random access operator `[]` as `arr[0]... arr[N-1]`. Indexing starts from zero!

Arrays are lvalues, they have storage and an address. However, objects of array type cannot be modified as a whole – they cannot appear on the left hand size of an assignment operator.

Example

```
int a[2] = {0,1};
int b[2];
b = a;    // error, invalid array assignment
```

However, due to the existence of an implicit copy-assignment operator

Example

```
struct s { int a[2]; } s = {1,2}, t;
t = s;
```

When arrays appear in context where arrays are not expected, but pointers are, this implicit conversion converts array type to a pointer to the first element of the array.

Example

```
void f(int (&ra)[3]) {}  // takes a reference to an array as argument
void g(int* pa) {}       // takes a pointer to element type as argument

int main()
{
    int a[3] = {1,2,3};
    int *p = a;
    cout << sizeof(a);  // 12 = 3 * sizeof(int) = 3 * 4
    cout << sizeof(p);  // 8, notice 64-bit architecture
    f(a);               // ok
    g(p);               // ok
    g(a);               // ok
}
```

An element type of an array may be an array type. In this case, such an array is called multi-dimensional.

Example

```
// an array of 2 arrays of 3 elements each
int a[2][3] = { {1,2,3}, {4,5,6} };
```

Such an array can be thought of as a 2×3 matrix.

Example

```
int a[2];              // an array of 2 integers
int b[2][3];           // an array of 2 arrays of 3 integers
int c[2][3][4];        // an array of two 3x4 matrices of integers
int* pa = a;           // a pointer to the first element in array a
int** pb = b;          // error, b is not implicitly converted to int**
int (*pb)[3] = b;      // b is converted to a pointer to the first row of b
int*** pc = c;         // error, c is not implicitly converted to int***
int (*pc)[3][4] = c;   // c is converted to a pointer to the first 3x4 matrix in c
```

If expr is ommitted, such an array is known as an **array of unknown bound**, which is sort of an incomplete type, with exception when used with an initializer.

Example

```
int a[];              // error, incomplete type
int a[] = {1,2,3};    // initializes an array of 3 integers
```

Multidimensional arrays cannot have an unknown bound other than the first.

Example

```
int a[][3];                          // error, incomplete type
int a[][3] = { {1,2,3}, {4,5,6} };   // the first dimension is 2
int a[2][] = { {1,2,3}, {4,5,6} };   // error, unbounded second dimension
```

Pointers and references to arrays of unknown bound can be created, but cannot be initialized or assigned with objects of known bound.

Example

```
extern int a[];
int (&ra)[] = a;
int (*pa)[] = &a;
int (*pa2)[2] = &a;  // error

int b[2] = {0,1};
int (&rb)[] = b;     // error
int (*pb)[] = b;     // error
int (&rb2)[2] = b;
int (*pb2)[2] = &b;
```

Pointers to arrays of unknown bound

- cannot participate in pointer arithmetic, but can be dereferenced
- cannot be used on the left of a random access operator []

References and pointers to arrays of unknown bound cannot be used as function arguments.

Functions

```
[attr] [modifier] T identifier ([argument list]) [cv] [ref] [except] [attr]
[attr] [modifier] auto identifier ([argument list]) [cv] [ref] [except] [attr] [-> T]
```

- [attr] any number of optional attributes
- T return type, cannot be a function type or array type, but can be pointer type or a reference to these types
- [arg] an optional list of function arguments
- [cv] optional cv-qualification, only allowed in non-static member function declarations
- [ref] optional ref-qualification, only allowed in non-static member function declarations
- [except] is dynamic exception specification or noexcept specification

Function modifiers may be a combination of

- explicit – the function cannot be used in implicit conversions and copy initialization
- static – a function with static or thread-local storage duration and internal linkage
- extern – a function with static or thread-local storage duration and external linkage
- thread_local – states that a function has thread-local storage
- constexpr – declares that it is possible to evaluate the value of the function at compile time

- inline – declares a function as inline, allows the compiler to substitute function body in-place of function calls.
- virtual – allows a class method to be dynamically bound.
- final – specifies that a virtual function cannot be overridden
- override – specifies that a virtual function overrides another virtual function (ref modifier)
- friend – grants a function access to private and protected members of the class where the friend declaration appears
- mutable – permits modification of the class member declared mutable even if the containing object is declared const

A function **declaration** bind a function **type** to a **name**.

Function **declaration** may appear in any scope. A function declaration at a class scope declares a function to be a member of a class (unless a friend specifier is used).

Non-member function **definition** may appear only in the namespace scope, member function definition may appear in class scope.

If auto is used as the return type, the trailing return type may be omitted and will be deduced by the compiler from the type of the returned expression.

Example

```
auto f() {
    int x = 2;
    return x;      // the return type is deduced to be int
}

const auto& g(int& x) {
    return x;     // the return type is deduced to be const int&
}
```

If there are multiple return statements, they must all deduce to the same return type, to avoid ambiguity.

Example

```
auto a(int x) {
    if (x > 0) return 2;
    else return 2.5;
}
```

If there are no return statements, the deduced type is void.

Example

```
auto a() {};   // the return type is deduced to be void
auto* b() {};  // error, only plain auto type can be deduced from void
auto& c() {};  // error, only plain auto type can be deduced from void
```

Once a `return` statement has been seen in a function, the return type deduced from that statement can be re-used in the same function.

Example

```
auto f(int x) {
    if (x > 0) return x;      // the return type is deduced to be int
    else return f(x+1);       // f's return type is already known
}
```

If the `return` statement uses brace-initialized list, the return type deduction is not allowed.

Example

```
auto f() { return {0,1}; }    // error
```

Function arguments may have default values in their declarations.

Example

```
int f(int a, int b=3) { return a + b; }  // argument b has default value 3
std::cout << f(5,5) << std::endl;         // will print 10 = 5 + 5
std::cout << f(5) << std::endl;           // f(5,3) is called, will print 8 = 5 + 3
```

Arguments with default values must appear in **the very last** position of a parameter list. In other words, all the arguments after the first argument with a default value, must have default values.

Example

```
int f(int a=3, int b) { return a + b; } // argument a has default value 3
f(5);    // ambiguity, is value 5 supplied as a value for a or b?
f(5,6);  // no ambiguity

int g(int a=3, int b=4) { return a + b; }
g();     // returns 7 = 3+4
g(2,2);  // returns 4 = 2 + 2
g(3);    // ambiguity, is a==3 or b==3?
```

In function **declaration**, the types of arguments are transformed according to the following rules.

1. Argument declarators are used to determine the type of argument.
2. If the type is an array type (array of T or array of unknown bound of T), it is converted to type pointer to T.
3. If the type is a function type T, it is converted to type pointer to T (pointer to function).
4. top-level cv-qualifiers are dropped from the parameter type

For this reason, `int f(int);` and `int f(const int);` declare the same function.

Example

The following function declarations are the same:

```
int f(int a[3]);
int f(int []);
int f(int* a);
int f(int* const);
int f(int* volatile);
```

Example

```
int f(int a);    // declaration
int g(int[10]); // declaration

// definition
int f(int const a) {
    return 0;
}

// definition
int g(int* p) {
    return 0;
}
```

Function arguments, as well as the return type of a function, cannot be incomplete types (i.e. declared but undefined).

A function body may be one of the following:

1. a compound statement (regular function body)
2. a function try block - associates a sequence of catch statements with the function body
3. explicitly deleted function definition =delete. Any use of a deleted function is ill-formed and produces compilation errors.
4. explicitly defaulted function definition =default

Example

An example of a function try block

```
int f(int a) try {
    return 0;
} catch (const std::exception& e) {
    // exception handler
} catch (...) {
    // exception handler
}
```

Example

Example of a deleted function

```
struct mytype {
    void* operator new(std::size_t) = delete;
    void* operator new[](std::size_t) = delete;
};

mytype* f = new mytype;       // error, use of deleted function new(std::size_t)
mytype* g = new mytype[10];   // error, use of deleted function new[](std::size_t)
```

A previously declared function cannot be redeclared as deleted, the deleted definition of a function must be the first declaration in a translation unit.

Example

An invalid definition

```
int f();                  // the first declaration of int f()
int f() = delete;         // deleted defintition of int f(), the second declaration
```

A valid definition

```
int f() = delete;         // the first declaration, deleted definition
```

Enumeration

An **enumeration** is a type whose values are restricted to pre-defined set of values.

Unscoped enumerations are of the form

```
enum [struct|class] name { enumerator[=constexpr], enumerator[=constexpr], ... }
enum [struct|class] name : type { enumerator[=constexpr], enumerator[=constexpr], ... }
enum [struct|class] name : type;
```

In the first type of declaration, the enumerator type is unspecified, it is an implementation-defined integral type.

In the second and third declarations, the enumerator type is fixed.

Each enumerator is associated with the value of underlying type.

If `struct` or `class` keywords are used, the enumeration is **unscoped**, otherwise it is **scoped**.

Example

Unscoped enumeration examples:

```
enum Color = { RED, GREEN, BLUE };
enum Level = { LOW, MEDIUM=10, HIGH=100 };
enum Level2 : char { LOW='l', MEDIUM='m', HIGH='h' };
Color c = RED;
Level l = LOW;
```

Example

Scoped enumeration examples:

```
enum struct Color { RED, GREEN, BLUE };
enum class Level { LOW, MEDIUM=10, HIGH=100 };
enum class Level2 : char { LOW='l', MEDIUM='m', HIGH='h' };
Color c = Color::RED;
Level l = Level::MEDIUM;
```

Smart Pointers

Smart pointers ensure that the managed dynamically allocated object is deleted when it is no longer used. 3 classes implement Smart Pointers:

1. std::unique_ptr - a smart pointer that uniquely owns dynamically allocated object and does not share this ownership.
2. std::shared_ptr - a smart pointer that shares ownership of a dynamically allocated object with other pointers.
3. std::weak_ptr - holds a non-owning (weak reference) to a dynamically allocated object.

A std::unique_ptr pointers are used for

1. providing exception safety to classes and functions by guaranteeing safe memory deallocation on normal and abnormal (i.e. exception) termination
2. passing ownership of uniquely-owned objects to functions
3. acquiring ownership of uniquely-owned objects from functions
4. as an element type in move-aware containers (i.e., std::vector)

An instance of std::unique_ptr may be constructed to manage an object of incomplete type. If the default deleter is used, the object type must be completed by the time when the deleter is invoked.

A unique pointer (std::unique_ptr) is a smart pointer that **owns** and **manages** another object, and **disposes** of the object when the unique_ptr goes out of scope.

The object is disposed of when

1. the unique_ptr object goes out of scope or is otherwise destroyed
2. the unique_ptr object is assigned another object via operator= or reset().

The object is disposed of using a **deleter**, which can be user-specified. The **default deleter** uses the delete call which deallocates the storage and destroys the object.

A unique_ptr may own no object. Such a pointer is called **empty**.

Example

```
std::unique_ptr<int> p;   // an empty pointer
p.reset(new int(30));     // now it manages a dynamic object
```

A unique_ptr may be initialized with a pointer upon creation.

Example

```
std::unique_ptr<int> p(new int(30));
```

Unique pointers are **NOT** copy-constructible nor copy-assignable.

Example

```
std::unique_ptr p(new int(30));
std::unique_ptr q(p);  // error, not copy−constructible
std::unique_prt r = p; // error, not copy−assignable
```

Unique pointers are, however, move-constructible and move-assignable.

Example

```
std::unique_ptr<int> p(new int(30));
std::unique_ptr<int> q(std::move(p));   // q is move−constructed, p is invalidated
std::unique_ptr<int> r = std::move(q);  // r is move−assigned, q is invalidated
```

Only non-const unique_ptr can transfer object ownership to another unique_ptr.

Example

```
std::unique_ptr<int> p(new int(30));              // ok
const std::unique_ptr<int> q(std::move(p));       // error
const std::unique_ptr<int> r = std::move(q);      // error
```

If the object is managed by a const-qualified std::unique_ptr, then its lifetime is limited by the scope in which the std::unique_ptr was created.

It is possible to specify a deleter function to std::unique_ptr.

Example

```
void deleter(int* p) {
    std::cout << "Custom Deleter" << std::endl;
    delete p;
}

int main() {
    std::unique_ptr<int,void (*)(int* ptr)> p(new int(30),deleter);
    return 0;
}
```

reset() allows to change the managed object. The previous object is deleted.

Example

```
std::unique_ptr p(new int(30));
p.reset(new int(50));           // memory storing value 30 is invalidated.
```

release() returns a raw pointer to the managed object and releases the ownership.

get() returns a raw pointer to the managed object (without releasing the ownership)

Example

```
std::unique_ptr p(new int(30));
int* raw = p.release();          // p contains no managed object
p.get();                         // returns nullptr, since there is no managed object
p.reset(new int(30));
p.get():                         // returns a non-zero address
```

The assignment operator (operator=) transfers ownership between two pointers.

Example

```
std::unique_ptr<int> p(new int(30));
p = std::unique_ptr<int>(new int(50));    // calls reset(src.release())
p = nullptr;                              // the same as p.reset(nullptr)
p = std::make_unique<int>(30);            // calls reset(src address)
```

The swap() method swaps the managed objects and their corresponding deleters between two instances of std::unique_ptr.

Example

```
std::unique_ptr<int> p(new int(30)), q(new int(50));
std::cout << *p << "\t" << *q << std::endl;     // 30 50
p.swap(q);
std::cout << *p << "\t" << *q << std::endl;     // 50 30
```

operator bool may be used to theck if std::unique_ptr has a managed object. Returns false, if the pointer is empty (no managed object).

Example

```
std::unique_ptr<int> p(new int(30));
std::unique_ptr<int> q;
// the following line prints "true false"
std::cout << std::boolalpha << p.operator bool() << "\t" << q.operator bool();
if (p) std::cout << "The value of p is " << *p << std::endl;  // is sent to stdout
if (q) std::cout << "The value of q is " << *q << std::endl;  // is not executed
```

Instances of std::unique_ptr have dereference operators *,-> and random access operator []. However, smart pointer do **NOT** support pointer arithmetic operators such as addition or subtraction.

A shared pointer (std::shared_ptr) is a reference counting smart pointer that shares ownership for an object with other shared pointers.

In other words, several shared_ptr instances may share the same managed object.

The managed object is destroyed and its storage is deallocated when the last shared pointer owning an object

1. goes out of scope or is otherwise destroyed
2. is assigned another managed object via operator= or reset()

Similar to unique pointers, shared pointers support custom deleter objects, can be empty (can have no managed object).

Differently from unique pointers, shared pointers are copy-constructible and copy-assignable.

Example

```
std::shared_ptr<int> s(new int(30));
std::shared_ptr<int> t = std::make_shared<int>(50); // copy−constructed, ok
std::shared_ptr<int> u(t);                 // copy−constructed, ok
std::shared_ptr<int> v = t;                // copy−assigned, ok
std::shared_ptr<int> x(std::move(t));    // move−constructed, ok
std::shared_ptr<int> y = std::move(v);   // move−assigned, ok
```

The use_count() method returns the number of references.

Example

```
std::shared_ptr<int> s(new int(30));
std::cout << s.use_count() << std::endl;    // prints 1
std::shared_ptr<int> t(s);
std::cout << s.use_count() << std::endl;    // prints 2
std::cout << t.use_count() << std::endl;    // prints 2
std::shared_ptr<int> u = t;
std::cout << s.use_count() << std::endl;    // prints 3
std::cout << t.use_count() << std::endl;    // prints 3
std::cout << u.use_count() << std::endl;    // prints 3
t = std::make_shared<int>(30);
std::cout << s.use_count() << std::endl;    // prints 2
std::cout << t.use_count() << std::endl;    // prints 1
std::cout << u.use_count() << std::endl;    // prints 2
```

When the counter goes down to zero (use_count() returns 0), the storage of the managed object is deallocated.

The managed object lives as long as there exist at least one shared pointer managing this storage.

unique() returns true if the object in question is managed only by the current shared_ptr instance.

Example

```
std::shared_ptr<int> s(new int(30));
std::cout << std::boolalpha << s.unique() << std::endl;     // true
std::shared_ptr<int> t(s);
std::cout << std::boolalpha << s.unique() << std::endl;     // false
std::cout << std::boolalpha << t.unique() << std::endl;     // false
```

owner_before() checks if a shared pointer preceeds the other w.r.t. the implementation-defined owner-based order. This ordering is used to allow shared pointers to be used as keys in associative containers (i.e. std::map).

Two shared pointers compare equivalent if and only if

1. they are both empty
2. they both own the same object

See the next slide for an example.

Example

```cpp
std::shared_ptr<int> m = nullptr;
std::shared_ptr<int> n = nullptr;
std::cout << std::boolalpha << m.owner_before(n) << std::endl;      // false
std::cout << std::boolalpha << n.owner_before(m) << std::endl;      // false
std::cout << std::boolalpha << (m == n) << std::endl;              // true

std::shared_ptr<int> s(new int(10));
std::shared_ptr<int> t(s);
std::cout << std::boolalpha << s.owner_before(t) << std::endl;      // false
std::cout << std::boolalpha << t.owner_before(s) << std::endl;      // false
std::cout << std::boolalpha << (s == t) << std::endl;              // true

std::shared_ptr<int> u(new int(20));
std::cout << std::boolalpha << s.owner_before(u) << std::endl;      // false
std::cout << std::boolalpha << u.owner_before(s) << std::endl;      // true
std::cout << std::boolalpha << (u == s) << std::endl;              // false
```

It is possible to create a new instance of std::shared_ptr obtained from another shared pointer using a cast expression on its stored raw pointer.

```
struct B {
    virtual ~B(){}
};

struct A : public B {
    ~A(){}
};

int main()
{
    std::shared_ptr<B> pb = std::make_shared<B>();
    auto pa = std::make_shared<A>();
    auto pb2 = std::static_pointer_cast<B>(pa);
    auto pa2 = std::dynamic_pointer_cast<A>(pb2);
    auto pa3 = std::const_pointer_cast<A>(pa);
}
```

Shared pointers share ownership of an object **only** by:

1. copy-constructing its value to another shared pointer
2. copy-assigning its value to another shared pointer

Constructing a new shared pointer from a raw pointer owned by another shared pointer is **undefined behavior**.

A shared pointer may point at a function

Example

```
void deleter(void(*)()) {}
void f() { std::cout << "f()" << std::endl; }
int main(){
    std::shared_ptr<void()> p(f, deleter);
    (*p)();
}
```

A shared pointer stores two pointers

1. the stored pointer (returned by `get()`)
2. a pointer to a control block, consisting of
   - A. a pointer to the managed storage, or the managed storage itself
   - B. allocator and deallocator functions
   - C. reference counter
   - D. number of weak pointers that refer to the managed object

The stored pointer is accessed by `get()` as well as the dereference and comparison operators. `reset()` replaces the managed object.

Shared pointers can share ownership of an object while pointing at another object (storing the address of another object) managed by another smart pointer.

Such smart pointers can be constructed from existing smart pointers using an aliasing constructor.

Example

```
int a = 2;
auto p = std::make_shared<int>(10);          // construct a "primary" pointer
auto q = std::shared_ptr<int>(p, &a);        // construct an aliased pointer
std::cout << p.use_count() << std::endl;     // prints 2
std::cout << q.use_count() << std::endl;     // prints 2
std::cout << p.get() << std::endl;           // points somewhere in the heap
std::cout << q.get() << std::endl;           // points somewhere in the stack
std::cout << *p.get() << std::endl;          // prints 10
std::cout << *q.get() << std::endl;          // prints 2
```

Following the same ideas, an empty shared pointer (without any managed object) may have a non-null stored pointer.

Example

```
int a = 2;
std::shared_ptr<int> p;                          // create an empty pointer
auto q = std::shared_ptr<int>(p, &a);            // create an aliased pointer
std::cout << p.use_count() << std::endl;         // prints 0
std::cout << p.get() << " "                      // prints 0
          << q.get() << " "                      // prints an address somewhere on the stack
          << *q.get() << std::endl;              // prints 2
```

An empty pointer with non-zero stored address is a non-owning shared pointer.

A weak pointer (`std::weak_ptr`) is a non-owning (weak) smart pointer that points at an object managed by some shared pointer. It can be used to access an object if it exists.

A weak pointer must be converted to shared pointer to obtain temporary ownership and to access the managed object.

If the main shared pointer goes out of scope, the lifetime of the managed object is extended until the temporary shared pointer is destroyed as well.

Similar to shared pointers, a weak pointer also stores two pointers:

1. a pointer to the control block
2. the stored pointer of the associated shared pointer (the one from which it was constructed)

To access the stored pointer, a weak pointer first must be locked. Locking means creating a temporary shared pointer that manages the object in question.

Example

```
void verifyObjectValidity(std::weak_ptr<int>& wp) {
    if (auto tsp = wp.lock()) {
        std::cout << "Object exists: " << *tsp << std::endl;
    } else {
        std::cout << "Object has expired" << std::endl;
    }
}

int main()
{
    std::weak_ptr<int> wp;
    {
        auto sp = std::make_shared<int>(30);
        wp = sp;
        verifyObjectValidity(wp);   // prints "Object exists: 30"
    }
    verifyObjectValidity(wp);       // prints "Object has expired"
}
```

Lambda expressions

There are 4 types of callables in the C++ language:

1. Functions
2. Pointers to functions
3. Functors (classes with the function call operator (`operator()`) defined)
4. Lambda expressions

You can think of a lambda as an unnamed (anonymous) inline function.

Like any function, a lambda

- represents a callable unit of code.
- has a return type
- has an argument list
- has a function body

Unlike any other function, lambda functions

- can capture entities from enclosing scope by value and by reference
- can be defined inside another function body
- cannot have default arguments

```
[[capture]][[(args)] [spec] [except] [attr] [->ret] { body }
```

- [capture] – an optional comma separated list of objects captured from the enclosing scope. If the lambda captures nothing, it has an empty capture list []
- [(args)] – an optional comma separated list of arguments
- [spec] – an optional set of specifiers: mutable,constexpr,consteval
- [except] – an optional exeption specification (throw(exception)) or a noexcept specification
- [attr] – any number of attributes (optional)
- [-> ret] – an optional trailing return type (used when the return type cannot be deduced from the return statement)

The meaning of the specifiers is as follows:

- mutable – allows body to modify parameters captured by value, as well as to call non-const member functions
- constexpr – states that the function call is a constexpr function – a function whose outcome can be deduced during compilation time.
- consteval – specifies that the function call is an immediate function – every call to the function must produce a compile time constant, implies inline modifier.

Note: consteval and constexpr modifiers cannot be used at the same time.

A consteval function is a constexpr function that is also inline.

Capture list allows to introduce captured entities from the enclosing scope into the inner scope of the functions.

[] means ”capture nothing”, the function does not use any local variables from an enclosing scope.

[a,b,c] means ”capture variables a,b,c by value”. Entities captured by value, are copied.

[&a,&b,&c] means ”capture variables a,b,c by reference”.

[=] means ”capture everything you can by value”

[&] means ”capture everything you can by reference”

[=, &a] means ”capture everything you can by value, but capture *a* by reference”

[&, a] means ”capture everything you can by reference, but capture *a* by value”

Capturing by reference does not extend the lifetimes of the captured entities. If the lambda call happens when the lifetime of the object, captured by reference, has ended, undefined behavior occurs.

Example

Define f as a callable object that takes no arguments and returns 42:

```
auto f = []{ return 42; };
```

We call a lambda as we would call any other function.

```
std::cout << f() << std::endl;
```

An omitted argument list it is inferred to an empty list.

The inferred return type depends on the function body

- If the function body contains a return statement, the return type is inferred from the type of expression that is returned.
- Otherwise the return type is void.

Example

Create a function *f* that compares two strings and returns true if the first string is shorter than the second one.

```
auto f = [](const std::string& a, const std::string& b) { return a.size() < b.size(); };
```

Example

Create a function *g* that returns `true` if a string, provided as argument, is longer than a given threshold `sz`.

```
std::size_t sz = 10;
auto g = [&sz](const std::string& str) { return str.size() > sz; };
```

Example

```
int val = 10;
auto f = [](int a, int b) { return a+b; };
auto g = [val](int a) -> bool { return a < val; };
auto h = [&val]() { val = 20; };

std::cout << f(10,10) << std::endl;     // prints 20
std::cout << std::boolalpha
          << g(5) << " "                // prints true
          << g(15) << std::endl;        // prints false
h(); std::cout << val << std::endl;     // prints 20
```

A function may return a callable object, which also means it is possible to return a lambda function from a function.

Example

```
std::function<int (int)> func() {
    return [](int x) { return x; };
}
```

Alternatively, a function can return an instance of a class that has a callable object as a data member.

If a function returns a lambda, the function must not return a reference to a local variable, and the lambda must not contain reference captures.

By default, a lambda may not change the value of the variable it captures by value.

Lambdas may change the value of variables captured by reference, if this reference is not const.

If we want to change the value of the variable captured by value, we must follow the lambda argument list with keyword mutable.

Lambdas marked as mutable may not omit the argument list, even if it is empty.

Example

What is the type of j? What is the value of j?

```
void func() {
    std::size_t v1 = 42;
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f();
}
```

Function Adapter

Function template std::bind is a general purpose function adapter.

It generates a new callable object from a given callable object, adjusting the argument list of the original callable object.

```
auto newCallable = std::bind(oldCallable,argList);
```

Example

```
#include <functional>

using std::placeholders::_1;
using std::placeholders::_2;

int diff(int a, int b) { return a - b; }
auto func = std::bind(diff,_1,10);      // func(int x) calls diff(x,10)
auto func2 = std::bind(diff,_2,_1);     // func(int x, int y) calls diff(y,x)

int main() {
    std::cout << func2(9,5) << std::endl;      // computes 5 - 9
    std::cout << func(12) << std::endl;        // computes 12 - 10
    return 0;
}
```

Example

```
#include <functional>

using std::placeholders::_1;

void f(std::string s, int n) { std::cout << "string " << s << "  " << "integer " << n
     << std::endl; }
auto f2 = std::bind(f,"Foo",5);      // f2() calls f("Foo",5)
auto f3 = std::bind(f,"Foo",_1);     // f2(int x) calls f("Foo",x)
auto f4 = std::bind(f,_1,100);       // f4(string s) calls f(s,100)

int main() {

    f2();                   // string Foo integer 5
    f2("Notfoo");           // string Foo integer 5
    f2("Notfoo",10);        // string Foo integer 5
    f3(10);                 // string Foo integer 10
    f4("Hello,World!"); // string Hello, World! integer 100
    return 0;
}
```

Argument to std::bind() that are not placeholders are copied into the callable object that std::bind() returns.

Sometimes we need to ass certain arguments to std::bind by reference (i.e., std::ostream& cannot be copied).

To pass an object to std::bind() without copying it, we can use function std::ref(). Similarly, the std::cref() function passes its argument by a constant reference into the callable.

Example

```
#include <functional>

using std::placeholders::_1;

void print(std::ostream& os, const std::string& msg) { os << msg << std::endl; }
auto wrapper = std::bind(print, std::ref(std::cout), _1);

int main()
{
    print(std::cout, "A message");
    wrapper("Another message");
    return 0;
}
```

Function Wrapper

Class template std::function is a polymorphic function wrapper that can store, copy, and invoke callable targets:

- functions
- lambda expression
- bind expression
- function objects
- pointers to member functions
- pointer to a data member (data member accessor)

std::function is copy-constructible and copy-assignable.

Example

std::function stores another function

```
void printNum(int a) { std::cout << a << std::endl; }
int main() {
    std::function<void(int)> func1 = printNum;
    func1(10);      // prints 10
    return 0;
}
```

Example

std::function stores a lambda expression

```
void printNum(int a) { std::cout << a << std::endl; }
int main() {
    std::function<void()> func2 = [](){ printNum(42); };
    func2();        // prints 42
    return 0;
}
```

Example

std::function storing a bind expression

```
void printNum(int a) { std::cout << a << std::endl; }
int main() {
    std::function<void()> func3 = std::bind(printNum,42);
    func3();         // prints 42
return 0;
}
```

Example

std::function storing a function object

```
void printNum(int a) { std::cout << a << std::endl; }
struct FObj {
    void operator()(int a) const {
        std::cout << a << std::endl;
    }
};
int main() {
    std::function<void(int)> func4 = FObj();
    func4(42);       // prints 42
    return 0;
}
```

Example

std::function stores a pointer to a member function

```
struct Foo {
    Foo(int num) { a = num; }
    int getNum() const { return a; }
    void setNum(int i) { a = i; }
    int a;
};
int main() {
    std::function<int(const Foo&)> f_getNum = &Foo::getNum;
    std::function<void(Foo&,int)> f_setNum = &Foo::setNum;
    const Foo foo(100);
    std::cout << f_getNum(foo) << std::endl;       // prints 100
    Foo foo2(200);
    std::cout << f_getNum(foo2) << std::endl;      // prints 200
    f_setNum(foo2,500);
    std::cout << f_getNum(foo2) << std::endl;      // prints 500
    return 0;
}
```

Example

std::function stores a data member accessor

```
struct Foo { int a; };

int main()
{
    const Foo foo{100};
    Foo foo2{200};
    std::function<int(Foo const&)> f_num = &Foo::a;
    std::cout << f_num(foo) << std::endl;       // prints 100
    std::cout << f_num(foo2) << std::endl;      // prints 200
    return 0;
}
```

Example

std::function stores a call to a member function and object

```
struct Foo {
    Foo(int x) { a = x; }
    bool compare(int x) { return a == x; }
    int a;
};

int main()
{
    Foo foo(100);
    using std::placeholders::_1;

    // passing a copy of instance foo
    std::function<bool(int)> f_cmp = std::bind( &Foo::compare, foo, _1 );
    std::cout << std::boolalpha << f_cmp(2) << std::endl;       // prints false
    std::cout << std::boolalpha << f_cmp(100) << std::endl;     // prints true

    // passing instance foo by reference
    std::function<bool(int)> f_cmp2 = std::bind( &Foo::compare, &foo, _1 );
    std::cout << std::boolalpha << f_cmp2(2) << std::endl;      // prints true
    std::cout << std::boolalpha << f_cmp2(100) << std::endl;    // prints false

    return 0;
}
```

Type Definitions

The typedef keyword creates type definitions.

Example

```
// simple type definition
typedef unsigned long ulong;

// the following objects have the same type
unsigned long l1;
ulong l2;

// more complex type definition
typedef int int_t, *intp_t, (&fp)(int,ulong), arr_t[10];

int_t a = 4;          // integer alias
intp_t pa = &a;       // pointer alias
fp x = foo;           // reference to a function int (&)(int,ulong)_
int a1[10];           // an array with 10 elements
arr_t a2;             // the same type as a1
std::cout << sizeof(a2) << std::endl; // prints 40

// the typedef keyword may be used anywhere in the specifier sequence
long unsigned typedef int long ullong;
```

Type Aliases

A type alias is a name that refers to a previously defined type (similar to typedef)

Example

```
using ulong = unsigned long;

// the following objects have the same type
unsigned long l1;
ulong l2;

using int_t = int;
int_t a = 4;

using intp_t = int*;
intp_t pa = &a;

using fp = int(&)(int,ulong);
fp x = foo;
```

?

THANK YOU
FOR
YOUR
ATTENTION
ANY QUESTIONS?