

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.

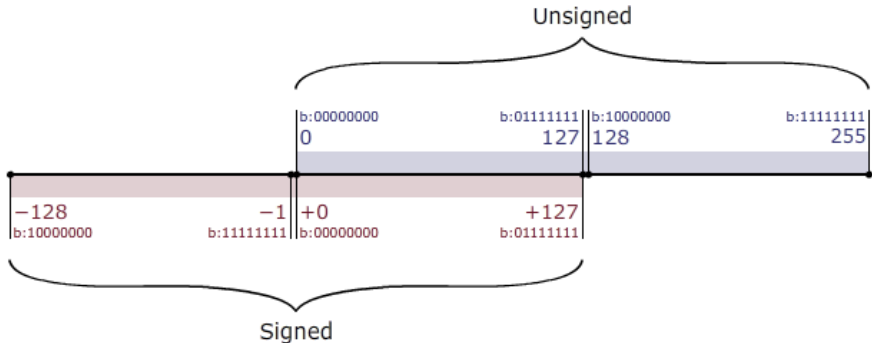
Common data models are the following:

- **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 signedness 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 signedness 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
```

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

```
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 lifetime 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 points 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, *vppl = &ll;
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 *>(vppl);
```

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 $k * \text{sizeof}(T)$ 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.



THANK YOU
FOR
YOUR
ATTENTION
ANY QUESTIONS?