
基础复习与扩展
Basics Review & Extension

现代C++基础 Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

Overview

- In this lesson, we'll mainly cover all things you (should) have learnt.
- Though mainly review, we'll also talk about some **miscellaneous but important** improvement in modern C++.
- So you may not skip this lesson.

- **Fundamental Types And Compound Types**
- **Expression**
- **Class**
- **Function Overloading**
- **Improvement in Execution Flow**
- **Template**

Basics

Fundamental Types and Compound Types

auto

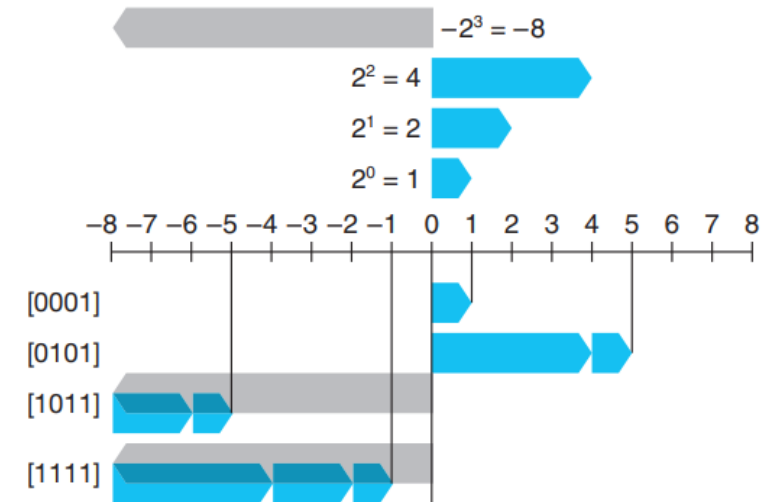
- Before all these types, let's introduce `auto` in C++11!
- You may substitute any type that can be deduced from the right side of `=` with `auto`!
 - E.g. `auto a = 1;` => `auto` is `int`
 - `auto` cannot deduce reference type and `const`, so you may need to specify them explicitly, e.g. `const auto& a = b;`.
 - To be exact, this is **type decay**, which will be discussed in the future.
 - It's also allowed to specify pointer, e.g. `auto* a = &b;`, though `auto a = &b;` is also correct.
 - Very useful for obvious but complex type, e.g. iterator.
 - We'll cover more usage of `auto` in the future...

Basics

- Fundamental types
 - Integer
 - Floating points
- Compound types
 - Pointer types & Reference types
 - Pointer-to-member types
 - Array types
 - Function types
 - Enumeration types
 - Class types

Integer

- In ICS, we've learnt the (widely accepted) binary representation of integers.
- Recap: Two's complement
 - This is the **de facto** standard of representing signed integers for all C++ compilers.
 - Range: $-2^{N-1} \sim 2^{N-1} - 1$ for signed values, $0 \sim 2^N$ for unsigned ones.
 - (unsigned) char-short-int-(long)-long long
 - Notice that **char** is not guaranteed to be signed or unsigned; if you hope to use exact signed one, you need to explicitly use **signed char**.
 - A single **unsigned** means **unsigned int**, but I recommend you to show **int** explicitly.



There are also many other character types, and we will cover them in the Unicode part.

Integer – some covert facts

- Note1: It's **UB** for **signed** integers to overflow.
 - E.g. $127 + 1$ for 8-bit signed integer.
 - Why?
 - Some architecture will trigger *Integer Overflow Exception* and it's not determined how OS will manage it.
 - E.g. [Some MIPS design](#), ADD and ADDU are different.
 - So what if an OS terminates the program?
 - In gcc, -ftrapv may help you to do so.
 - But it's defined for **unsigned** ones, just rounded back (i.e. $\text{mod } 2^N$).

Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

ADD

Add Word

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000				rs	rt		rd		0 00000		ADD 100000
6				5	5		5		5		6

Format: ADD rd, rs, rt

Purpose: Add Word
To add 32-bit integers. If an overflow occurs, then trap.

Description: $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$
The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is sign-extended and placed into GPR *rd*.

Restrictions:
If either GPR *rs* or GPR *rt* does not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is UNPREDICTABLE.

Operation:

```
if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
  UNPREDICTABLE
endif
temp ← (GPR[rs] << 31 | GPR[rs][31..0]) + (GPR[rt] << 31 | GPR[rt][31..0])
if temp > 0xFFFFFFFF then
  SignalException(IntegerOverflow)
else
  GPR[rd] ← sign_extend(temp << 31..0)
endif
```

Exceptions:
Integer Overflow

Programming Notes:
ADDU performs the same arithmetic operation but does not trap on overflow.

You may check more information for integer overflow exception in MIPS [here](#).

- IMPORTANT: unsigned integer is always ≥ 0 !
 - So loop like `for(unsigned int a = xx; a \geq 0; a--)` is an infinite loop!
 - There are many hidden bugs related to this, e.g.
 - This is because `std::string::find` will return `std::string::npos`.
 - This value is -1, but in fact `size_type` (basically just `size_t`).
 - We'll cover `size_t` in *Container*, but now we just need to know it's **unsigned**.
 - Then the first branch is always taken!
 - Solution: usually we use `!= std::string::npos`.
 - Since C++20, `std::cmp_XXX` defined in `<utility>` can safely compare unsigned and signed integers (excluding `bool`/characters).
 - `std::cmp_greater_equal(std::string::npos, 0)` will return `false`.
 - `std::in_range<T>(x)` can also be used to check whether a value is representable by integer type `T`.
 - Anyway, you need to be careful with operations in unsigned value.

你见过最烂的代码长什么样子?



天苍苍

+ 关注

25 人赞同了该回答

```
1 #include<iostream>
2 int main()
3 {
4
5     std::string s = "hello world";
6     if (s.find("t")>=0)
7     {
8         std::cout << "found\n";
9     }
10    else
11    {
12        std::cout << "not found\n";
13    }
14
15
16 }
```

Microsoft Visual Studio 调试

found

F:\projects\CPP_PROJECTS\CppTest\x64\Debug\CppTest.exe (进程 58788) 已退要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时按任意键关闭此窗口”

我之前遇到一个问题, 这是代码简化版本, 看大家能理解么

Integer – some covert facts

- Note2: It's **UB** for integers to divide 0.
- Note3: All arithmetic operations will promote integers that are smaller than `int` to `int` first.
 - i.e. (unsigned) `char/short` will be converted to `int` first, and only then the arithmetic operations will be done next.
 - *Or unsigned int if int cannot represent them, in case `sizeof(short) == sizeof(int)`.

• E.g.

```
#include <iostream>

int main()
{
    unsigned short s1 = 0xFF00, s2 = 0x100, s3 = 0xFFFF;
    if (s1 + s2 > s3)
        std::cout << "Unexpected!\n";

    unsigned int i1 = 0xFFFF0000, i2 = 0x10000, i3 = 0xFFFFFFFF;
    if(i1 + i2 > i3)
        std::cout << "Unexpected, too!\n";

    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

Unexpected!
请按任意键继续. . .

Integer

- So what's the size of integers?
 - Not strictly regulated:
 - `1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
 - 1 is almost always 1 byte but may also be other things like 4 bytes (as some DSPs do, damn).
 - As in bits, `8 <= char; 16 <= short, int; 32 <= long; 64 <= long long`
 - But there are some widely used data models:
 - ILP32(4/4/4): int, long, pointers are 4 bytes.
 - Widely used in 32-bit system.
 - LLP64(4/4/8): int, long are 4 bytes; pointers are 8 bytes.
 - Widely used in Windows.
 - LP64(4/8/8): int are 4 bytes; long and pointers are 8 bytes.
 - Widely used in Linux and MacOS.
- 1 byte == 8 bits (de facto **now**, but some rare arch may forbid grasping only a single byte, so you may also say the smallest group is a byte there).

Integer

- You may observe heavy load for C++ to consider all hardware and architecture cases.
- So can we make it cross-platform?
 - E.g. `long` is a very stupid choice since it's 4 bytes in Windows and 8 bytes in Linux.
- In `<cstdint>`, there are some fixed width integers:
- `std::int(x)_t`, `std::uint(x)_t`, where $x=8/16/32/64$
 - However, considering that some platforms may not support e.g. 8 bits, these are **all optional**.
 - Nevertheless, they're supported in most of platforms, and using them also denotes that you don't want the program used in other weird platforms (causing compile error rather than unexpected runtime error).

Integer

- If you really want to write very general code, you may use:
 - `std::(u)int_least(x)_t`, where $x=8/16/32/64$, meaning that the integer type has the smallest size that is bigger than x bits.
 - E.g. if a system supports 16/64 bits, `int_least8_t` has 16 bits.
 - `std::(u)int_fast(x)_t`, where $x=8/16/32/64$, meaning that the integer type is the integer that works fastest in this arch and is bigger than x bits.
 - `std::(u)intmax_t`, the biggest integer that is supported by the system.
- Additional Note: these types are only an **alias** of `int/long long...`, not a different type!
 - In usual cases, `std::uint8_t` is `unsigned char`, so if you hope to output the address of the `std::uint8_t`, it will try to output a C-string (i.e. output content until null-termination), which is totally unexpected!
 - You need to convert `std::uint8_t*` to `void*` to get the address.

Bit Manipulation of Integers

- In the data lab of ICS, we struggle to play tricks on integers.
 - Is there any way to utilize the standard library?
 - In C++20, `<bit>` is provided to do so!
 - About the power of 2 (all accept one parameter, i.e. the integer):
 - `std::has_single_bit`, check whether an integer has only a single 1 in bits.
 - `std::bit_ceil`, get the smallest integer that is power of two and not less than the given value.
 - i.e. if an integer `has_single_bit`, do nothing; else set the next bit of the highest 1 and clear all other bits.
 - If the highest bit has been set (i.e. result is not representable), UB.
 - The width can be got by `std::bit_width`, which returns $1 + \lfloor \log_2 x \rfloor$.
 - `std::bit_floor`, get the biggest integer that is power of two and not greater than the given value.
 - For 0, return 0.
- All functions in `<bit>` are for **unsigned** integers; if you want to use in signed ones, you need to convert them to unsigned ones.

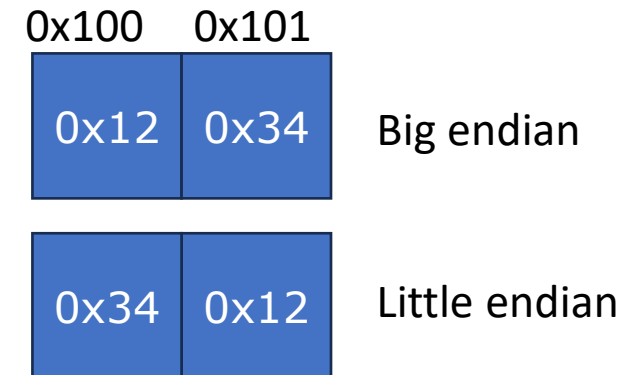
Bit Manipulation of Integers

- For bit rotation (negative shift is also accepted):
 - `std::rotl(x, int shiftBits)`: left rotate
 - e.g. shift 2 bits: 00011101->01110100
 - `std::rotr(x, int shiftBits)`: right rotate
 - e.g. shift 3 bits: 00011101->10100011
- For counting (return `int`):
 - `std::count(l/r)_(zero/one)`, count the number of consecutive 0/1 bits starting from left/right.
 - E.g. 12=00001100, `std::countl_zero` is 4, `std::countr_zero` is 2.
 - `std::pop_count`, count the number of 1 in an integer.
 - E.g. 24 has 2 bits.
- We don't dig into how they are implemented (you may already learn them in the review of data lab).

If you're interested in bit tricks, I recommend *Hacker's Delight*, 2nd ed. by Henry S. Warren, Jr. (中译名: 算法心得-高效算法的奥秘)

Bit Manipulation of Integers

- Finally, since C++20, you can use `std::endian` to check the endianness of platform.
 - `std::endian::little/big/native`, where `native` is the endianness of current platform.
 - If the current platform is mixed endian, then `native != little && native != big`.
 - For example, 0x1234:
 - i.e. little endian should be viewed from big address **to little address** to get correct value (like a stack).
- Since C++23, you can use `std::byteswap` to swap an integer byte by byte.
 - i.e. reverse the endianness.



Basics

- Fundamental types
 - Integer
 - Floating points
- Compound types
 - Pointer types & Reference types
 - Pointer-to-member types
 - Array types
 - Function types
 - Enumeration types
 - Class types

Floating point

- IEEE 754: sign bit + exponent field + fraction field
 - Fraction field $0.x_1...x_n = \sum 2^{-ix_i} = M$
 - Bias = $2^k - 1$, where k is the length of the exponent field.
 - Denormalized value: exponent field = 0, value = $M \cdot 2^{1-Bias}$
 - Normalized value: value = $(1 + M) \cdot 2^{exp-Bias}$
 - Special value: exponent field = 11...11
 - If fraction field is 0, **inf**
 - Else **NaN**.
- Round-to-even/zero/down/up
- Any comparison except for **!=** is false for **NaN**; **&&/|| NaN** is true;
Any arithmetic operations get **NaN**; **inf+(-inf)= NaN**
- **+inf** is bigger than all non-special values; **-inf** is smallest.

Floating point

- `float`: 23 fraction + 8 exponent;
`double`: 52 fraction + 11 exponent
- Corollary: For normalized numbers of `float`, plus and minus will have effects iff. their magnitude is in $2^{24} \approx 1.6e^7$.
 - This magnitude is so great that we may usually leave it off.
 - But is it that “great”?
 - Cornell Box: The walls and light is just a plane, so the bounding box will get intersection suddenly(i.e. enter and leave “at the same time”).
 - Intersection law in graphics: $t_{enter} < t_{leave} \ \&\& \ t_{leave} > 0$.
 - But considering zero height, $t_{enter} \leq t_{leave}$.
 - What about $t_{enter} < t_{leave} + \epsilon$? It seems that they are equivalent.

Notice that float/double isn't regulated to be IEEE754, in case you're using some weird platform.



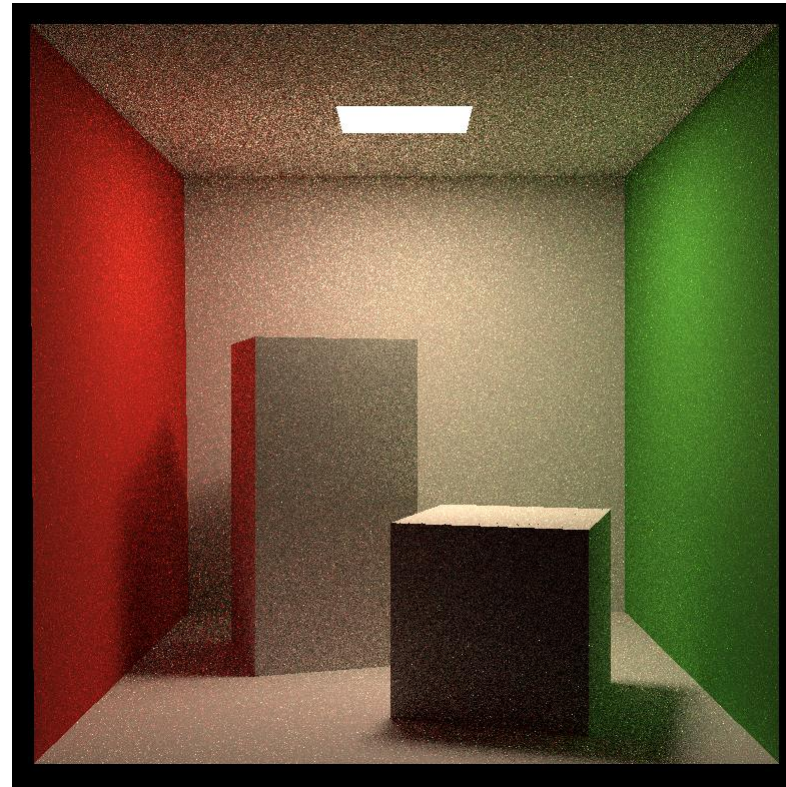
Floating point

$\epsilon = 1e^{-5}$, scene is about $560 * 560$.

- $t_{enter} < t_{leave} + \epsilon$



- $t_{enter} \leq t_{leave}$



SPP=16, g++ -O3, no multi-threading and GPU, 684 seconds

Floating point

- This bug bothers me for about 8 hours.
- During debugging, I print the distance between the origin and the intersection point, leading to about 300~800.
- Notice that $\epsilon = 1e^{-5} \Rightarrow magnitude = 3e^7 \sim 8e^7 > 1.6e^7$.
- So now $t_{enter} < t_{leave} + \epsilon \nRightarrow t_{enter} \leq t_{leave}$
- So unrepresentable is usually unexpected, is there any way to prevent that?
 - For integers, you may use [SafeInt](#) library; only disable it in release mode.
 - For floating point, you may also wrap a utility class.

Floating point

- **long double**: 128bit(112+15)/80bit(63+15)/64bit(msvc)
- In C++23, floating points are **optionally** extended, just like fixed-width integers.
 - **NOTICE**: MS seems not ready to implement this feature.
- Defined in **<stdfloat>**
 - **std::float16_t**: 10+5
 - **std::float32_t**: 23+8
 - **std::float64_t**: 52+11
 - **std::float128_t**: 112+15
 - **std::bfloat16_t**: 7+8
- They are all **different types** rather than alias!

Used in deep learning, expand the range and lower the precision (weights don't need high precision; even 4-bit integer is used in some design).

Bit Manipulation for Floating Point

- We've mentioned that functions in `<bit>` are all for unsigned integers.
- So how can we utilize them in floating points?
- `std::bit_cast<ToType>(fromVal)` will help you to do so.
 - Cast a value to another type with the same bit contents.
 - Only if `sizeof(fromType) == sizeof(toType)` can it succeed.

```
unsigned int AddOneInExp(unsigned int a)
{
    const int fracBits = 23, expBits = 8, fracMask = (1 << fracBits) - 1,
            expMask = ((1 << expBits) - 1) << fracBits;
    unsigned int exp = a & expMask;
    unsigned int clearNum = (a & (~expMask));
    return clearNum | (exp + (1 << fracBits));
}
```

```
int main()
{
    float f1 = 1.0;
    unsigned int i1 = std::bit_cast<unsigned int>(f1);
    unsigned int i2 = AddOneInExp(i1);
    float f2 = std::bit_cast<float>(i2);
    std::cout << std::format("f1 = {}, i1 = {:032b}, i2 = {:032b}, f2 = {}\n",
        f1, i1, i2, f2);
    return 0;
}
```

Or `std::println()`

f1 = 1, i1 = 00111111100000000000000000000000, i2 = 01000000000000000000000000000000, f2 = 2
请按任意键继续. . .

Literal Prefix and Suffix

- For literals, prefix is used to identify format, and suffix is used to identify type.
- 1 – int; 1l/1L – long; 1ll/1LL-long long
 - Add a u/U will make them unsigned, e.g. 1u, 1ull, 1llu
- 10 – decimal; 0x10 – hexadecimal (get 16); **010 – octal** (get 8)
- Since C++14
 - Binary is supported, i.e. 0b10 (get 2).
 - Separator is supported, i.e. **0x11FF**' 3344; **0b0011**'0100
 - E.g. I use it in RISC-V simulator in decoding mask.

Literal Prefix and Suffix

- For floating point, **1e-5** means 10^{-5} .
- 1.0 – double; 1.0f – float; 1.0L – long double
 - (f/F)16/32/64/128; bf16/BF16
- .1 means 0.1 (zero before fraction can be omitted).

C++17 supports hex float, but we don't dig into that.

Final word

- If you want to get special values for an arithmetic type, you may use `<limits>`.
 - E.g. `std::numeric_limits<int>::max()` get 0x7FFFFFFF (assuming 32 bit).
 - `std::numeric_limits<float>::max()` get the maximum **finite** number.
 - There are lots of other limits like `denorm_min`, and we don't dig into them (just search it when you really need).
 - You may also use things like `INT32_MAX` defined in `<cstdint>`; it's more like a C-style way.
- `bool`: special integer, only `true` (non-zero, not definitely 1) or `false`. But convert it to other integer types will get 1/0.
 - `sizeof(bool)` is not necessarily 1.
 - `bool` is not allowed to `++/--` since C++17.

Basics

- Fundamental types
 - Integer
 - Floating points
- Compound types
 - Pointer types & Reference types
 - Pointer-to-member types
 - Array types
 - Function types
 - Enumeration types
 - Class types

Pointer Type

- Pointer can be seen as address of an object.
- By `&`, we can get the pointer; by `*`, we can dereference the pointer, e.g. `T b; T* a = &b, *d = &b; T c = *a;`
- However, it's not correct to just see it as an address and drop its type. You may not cast it to another pointer type and dereference it in many cases, which will be covered in the lecture of *Type Safety*.
- Special value - `NULL`; you may use `nullptr` in C++11.
 - It's UB to dereference null pointer (but usually terminate the program).
 - `nullptr` is also introduced in C23.

Null pointer

- Why should we use `nullptr` instead of `NULL`?
- In C++, `NULL` is just an integer rather than a pointer!
- So what will this program output?

```
#include <iostream>

void Nope(int a) { std::cout << "Int!\n"; }
void Nope(void* a) { std::cout << "Pointer!\n"; }

int main()
{
    Nope(NULL);
    return 0;
}
```

```
#ifndef NULL
#ifdef __cplusplus
#define NULL 0
...
#else
#define NULL ((void *)0)
#endif
#endif
```

vcruntime.h

```
C:\WINDOWS\system32\cmd.exe
Int!
请按任意键继续. . .
```

- Really unexpected!

Reference Type

- In some cases in C++, we can use reference as a syntax sugar for pointer.
 - E.g. we may want to set a variable out of scope, then pointer can be passed.
 - Reference can omit taking address, which may be easier to understand in some cases.
- Differences:
 - Reference is not nullable.
 - Reference cannot change the referred object.
 - Reference may or may not occupy any memory, while pointer will definitely.
 - But this isn't usually that important because of compiler optimization.
 - Reference is forbidden to be template parameters in many parts of standard library, like you cannot `std::vector<A&>`.

cv-qualifier

- **const** and **volatile**.
 - We've learn in ECF in ICS that **volatile** is used to force the compiler to always get the content from memory.
 - Otherwise, it may be optimized to load from registers, and always operate the register.
 - This is used when the exception handler changes some variables. The function **never knows** that it will be changed since OS exceptions are unexpected, so if you read it twice, the compiler may optimize it as reading only once from memory and another from register.
 - There are also other cases that you may change the memory of a process while it doesn't know, then you must use **volatile** to refresh.
 - **const**: label something as unchangeable.

Basics

- Fundamental types
 - Integer
 - Floating points
- Compound types
 - Pointer types & Reference types
 - Pointer-to-member types (We may cover it in the future)
 - Array types
 - Function types
 - Enumeration types
 - Class types

Array types

- E.g. `int a[3] = {1,2,3}, b[] = {4,5,6};`
 - `3` can be omitted since the size can be deduced.
 - If provided elements are less than given size, the rest will be 0.
 - However, if no elements are provided, array on stack will have random values.
 - Though you may see array as pointer before, it's not true...
 - That's in fact *decay*; we'll also cover it in *Type Safety*.
 - A special array is C-string, e.g. `const char* str = "test";`
 - `"test"` is `const char[5]`, but it may decay to `const char*`.
 - `char*` (drop `const` for string literals) here is not permitted in C++.
 - Null-terminated, so different with `char` array itself.
 - There are no array of functions or array of references.
 - VLA (i.e. `int arr[n]`) is not allowed in C++.

Array types

- Multidimensional array
 - E.g. `int a[2][3] = { {1,2,3},{4,5,6}};`
 - All dimensions but the first should have explicit size.
- Dynamic allocation
 - We've learnt `malloc` in C (function defined in `<stdlib.h>`) and `new/new[]` in C++ (as a keyword).
 - In ICS, we know that they are allocated in heap.
 - Difference: `malloc` will only allocate memory, `new` will call ctor if the object provides one.
 - It's usually preferred to use `new` in C++.
 - If you just want memory without constructed objects, you may use allocator in C++, which will be covered in *Memory Management*.

Dynamic Allocation

- Let me show you how to allocate space in heap.
 - This is rarely used in fact, just for the purpose of review.
- Notice that `new` and `malloc` will get a pointer instead of an array!
 - It points to the content of some continuous space.
- When memory is exhausted, `malloc` will return `NULL`.
 - `new` will *throw* `std::bad_array_new_length`, an *exception* inherited from `std::bad_alloc`, defined in `<new>`.
 - We'll cover exception in *Error Handling* and more things about `new` in *Memory Management*.
- Deallocation: `free` and `delete/delete[]`.
 - They are not interchangeable; `delete` will call dtor of objects.
 - Deallocation will & should always not throw exception.

Basics

- Fundamental types
 - Integer
 - Floating points
- Compound types
 - Pointer types & Reference types
 - Pointer-to-member types
 - Array types
 - **Function types**
 - Enumeration types
 - Class types

Function

- To reduce the duplication of code and increase the modularity, functions are important components.
- Function prototype (declaration):

```
void Foo(int = 1);
```

 - You may provide default parameters.
 - Semicolon is necessary; param name is not.
- Function implementation (definition):

```
void Foo(int param) {  
    ...  
}
```

 - Param types and return type should be same as prototype; default params should not appear here.
 - The definition can specify return type as **auto** since C++14; it can be deduced from **return** statement (but **ref** and **const** is still not deduced.).
 - If types of multiple **return** differ, compile error.
- You may also define function definition directly; then default params should be provided there if any.

We'll cover why you need function prototypes in *Programming with Multiple Files*.

Function

- The return type cannot be function or array (but can be their references or pointers).
 - However, you shouldn't return references (and also pointers) of **local variables**, since they will be destroyed after exiting the function, which makes them **dangling** references/pointers.
- **static** local variable: "global variable" in the function, not directly accessible to others.
 - Initialized iff. the first time its initialization is executed.
- Function pointers
 - It's quite like function prototype:
 - The type is **void (*)(int)**, and **FooPtr** is the name.
 - Ampersand can be omitted; The function type is **void(int)** and in fact cannot be used to **define** a named variable, but it can **decay** to pointer implicitly.

```
int square() {  
    static int i = 0;  
    ++i;  
    return i * i;  
}
```

```
void (*FooPtr)(int) = &Foo;
```

Understanding complex types...

- When the return type is function pointer, the prototype is like:
 - Oops, WTF `int Bar(int) { return 0; }` `int (*Foo(float realParam))(int) { return &Bar; }`
- When **const** is added in functions, pointers and arrays, it'll be more complex to understand the meaning...
 - **const int ***: pointer to **int**, and **int** is **const**; meaning that ***p = 1;** is forbidden. This is same as **int const ***.
 - **int * const**: the pointer is **const**, and points to **int**. meaning that **p = &a;** is forbidden.
- Generally, **clockwise/spiral rule**

Clockwise/Spiral Rule*

- Notice that this is to make you understand some smelly or legacy code, and we'll teach the good way later.
- Procedures:
 - Start from unknown symbol.
 - Move in clockwise/spiral direction, and when encountering
 - [X]/[]: array
 - (type, ...): function
 - *: pointer to...
 - E.g. A pointer to a function, which returns `char*` and accepts `int + float*`

```
+-----+
| +---+ |
| |++|  |
| |^|   |
char *(*fp)( int, float *);
^   ^ ^  |
|   | +---+ |
|   +-----+
+-----+
```




吴咏炜

编程四十年, <http://wyw.dcweb.cn/>

Clockwise/

你经常看 TA 的内容

C的数组语法跟其他复杂特性一旦混合就是一坨shit, 可读性非常糟糕。想办法规避, 而不是(如其他回答教的那样) 用正规写法。

- E.g. `int(*Foo(int, int(*)(int)))(int*)(float);`
 - `Foo` -> (, meaning that `Foo` is a function.
 - The function parameters are `int` and `int(*)(int)`;
 - We can use spiral rule to resolve it; we know it's a function pointer.
 - Return type is...
 - `(->*`, meaning that the function returns a pointer that points to...
 - `*->`(, meaning that the pointer points to a function.
 - `(->int`, meaning that the function returns `int`.
 - `int->int(*)(float)`, we can also use spiral rule to resolve it.
 - We know it means it accepts another function pointer.
 - What the hell, too disgusting...
 - Here we even omit many other obscure legal ways from C!
 - The ploughman homeward plods his weary way,
And leaves the world to darkness and to me. --- *Elegy Written in a Country Churchyard*,
Thomas Gray

Type alias

May be misleading,
actually pointer type.

- **using** in C++11 will save you a lot of trouble.
- Cool, clean, clear!
- It creates an alias of the type, so it's the same type as before except for the name.
- In C, you may use **typedef**, but it's not as intuitive & powerful as **using**.
 - We'll cover more usage of **using** other than this simple type aliasing in the future.

```
using MyFuncType1 = int (*)(float);  
using MyFuncType2 = int (*)(int);  
using MyFuncType3 = int (*)(MyFuncType1);  
MyFuncType3 Foo2(MyFuncType2 func) {
```

```
typedef int (*MyFuncType1)(float);  
typedef int (*MyFuncType2)(int);  
typedef int (*MyFuncType3)(MyFuncType1);  
typedef int Type;  
using Type2 = int;
```

Particularly, it's not same as pure text replacement! You see the alias **as a whole**. For example, **using CPtr = char***; **using ConstCPtr = const CPtr** will not get **const char***, but **char* const** i.e. the pointer itself cannot be changed.

Attribute

- Sometimes, the return value of a function should not be omitted, but there is no way to force users to notice that...
 - E.g. The result of `+`, usually a single `a + b`; without assigning is weird.
- Since C++11, `[[attribute]]` is introduced; this is a standard way to extend the language.
 - E.g. `[[omp::directive()]]` for OpenMP
 - C++ itself also introduces some basic attributes that are widely extended in many compilers.
- For function, you may use `[[nodiscard]]` since C++17 **before return type** to denote the return value should not be dropped.
 - Since C++20, you may specify reason as `[[nodiscard("reason")]]`.
 - The compiler will report a warning if it's dropped.

C++

In C++11 and higher, all OpenMP directives may be specified with C++ attribute specifiers as follows:

```
[[ omp :: directive( directive-name[, clause[, clause]... ] ) ]]
```

Attribute

- It's recommended to specify the attribute both in declaration and in definition.
- There are also other attributes for function:
 - `[[deprecated]]` and `[[deprecated("reason")]]` since C++14: mark the function as deprecated so that users should not use it in new code.
 - This can also be used in namespaces and enumerators.
 - `[[noreturn]]` since C++11: some functions may never return e.g. `fatal()` here, but if you don't return something in `test()`, the compiler may give you a warning. Specify `fatal` with this attribute will suppress the warning, and also tell users that this function will never return.
 - `[[maybe_unused]]` since C++17: when some entities (functions, variables including params) seem unused, compilers may report a warning. This will suppress the warning.
 - "seem" is because you may do something inner, e.g. for creating singleton.
 - Or, you may want to match some function type to unify pointer, e.g. in Pintos.

```
void fatal(){
    std::exit(1);
}
int test(bool flag)
{
    if (flag)
        return 1;
    fatal();
    // warn: no return.
}
```

```
using thread_func = void (*)(void* aux);    void idle_thread([[maybe_unused]] void* aux)
```

Basics

- Fundamental types
 - Integer
 - Floating points
- Compound types
 - Pointer types & Reference types
 - Pointer-to-member types
 - Array types
 - Function types
 - Enumeration types
 - Class types

Enumeration

- Enumeration is a type whose values are restricted to several options.

- E.g.

Days in a week:

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };  
Day day = Monday;
```

- They are actually integers; By default, the first element starts from 0, and the second is 1, etc.

- You may also designate some value, e.g.

```
enum Day { Monday, Tuesday, Wednesday = 10, Thursday, Friday, Saturday = 20, Sunday };
```

- Then **Thursday** is 11, **Friday** is 12, **Sunday** is 21.
 - If you make **Saturday = 10**, then **Saturday == Wednesday** and **Thursday == Sunday**.
 - You can do any integer operations on them, e.g. +/-/...

Enumeration

- You may notice that using values in is just like using global variables (“unscoped”), which pollutes normal variable names that can be used by programmers.
 - It may also be dangerous to perform unexpected integer operations.
- Since C++11, scoped enumeration is introduced to enhance the security.

```
enum class Day { Monday, Tuesday, Wednesday = 10, Thursday, Friday, Saturday = 20, Sunday };  
Day day = Day::Monday;
```

- **Day::** is required to get the value, which reduces pollution in the current “space”.
- Only comparisons are permitted; arithmetic operations will trigger compile error.

Enumeration

- If you really want to do integer operations, you need to convert it **explicitly**.
 - You may use `(int)Day::Monday` currently, which will be improved in *Type Safety*.
 - By default, the underlying type is `int`; you may also change it to other integer types:

```
enum class Day : std::uint8_t { Monday,
```
 - You may use `std::underlying_type<Day>::type` or `std::underlying_type_t<Day>` (since C++14) to get the integer type (e.g. here is `std::uint8_t`); defined in `<type_traits>`.
 - In C++23, you may also use `std::to_underlying<Day>(day)` to get the underlying integer directly; defined in `<utility>`.
 - However, it's still legal to **initialize** like `Day day{1};` (since C++17); But it's illegal to use `Day day = 1;` or assign `day = 1`.
 - It keeps the underlying value, comparing enumerations are same as comparing underlying values..
 - We'll tell you the difference of these two initialization later.

Enumeration

```
enum access_t { read = 1, write = 2, exec = 4 }; // enumerators: 1, 2, 4 range: 0..7
access_t rwe = static_cast<access_t>(7);
assert((rwe & read) && (rwe & write) && (rwe & exec));

access_t x = static_cast<access_t>(8.0); // undefined behavior since CWG 1766
access_t y = static_cast<access_t>(8);   // undefined behavior since CWG 1766

enum foo { a = 0, b = UINT_MAX }; // range: [0, UINT_MAX]
foo x = foo(-1); // undefined behavior since CWG 1766,
                // even if foo's underlying type is unsigned int
```

- Enumeration is also widely used in bitwise operation.

- `|` means “enable the option”; `&` means “check the option”.
- This is used in some external libraries...

```
const aiScene* model = importer.ReadFile(modelPath.string(),
    aiProcess_Triangulate | aiProcess_GenSmoothNormals | aiProcess_FlipUVs
    | aiProcess_JoinIdenticalVertices);
```

- As well as in the standard library:

```
std::fstream f{ "test.txt", std::ios::out | std::ios::in };
```

- However, scoped enumeration doesn’t support arithmetic operations, so you may either use unscoped one carefully (e.g. with `namespace`; we’ll cover it later) or define the operator yourself (we’ll show you in *operator overloading* later.).
- Final word: you need to ensure not to exceed the limit of enumeration value in the meaning of bits (i.e. $(1 \ll \text{std::bitwidth}(\text{MaxEnum})) - 1$ or $(1 \ll (\text{MSB}(\text{MaxEnum}) + 1)) - 1$), otherwise UB.

Basics

- Fundamental types
 - Integer
 - Floating points
- Compound types
 - Pointer types & Reference types
 - Pointer-to-member types
 - Array types
 - Function types
 - Enumeration types
 - Class types (Before that, let's talk about expression...)

Basics

Expression

Expression

- You may think: what? Why is expression reviewed? That's too easy!
 - Well, it should be, but many C++ programmers (including professionals) have huge misunderstanding on it...
- There are three basic concepts in operators & expression evaluation:
 - Precedence
 - Associativity
 - Order
- You may check precedence and associativity in any basic C++ book or wiki, e.g. [here](#).

Expression

- From the view of compiler (you'll also learn it in *Compiler Principles*), an expression is in fact a tree, determined by associativity and precedence.
 - Precedence is used to split terms first, e.g. $9/3/2+2*5$ is split to `root(+)->leftChild(9/3/2)->rightChild(2*5)`, because `*`, `/` has higher precedence.
 - Associativity determines how the tree will grow.
 - Left-to-right associativity will make it grow on the left, while right-to-left is on right.
 - E.g. `/` has left-to-right associativity, so $9/3/2$ is actually `root(/)->leftChild(9/3)->rightChild(2)`.
 - For example, `/=` has right-to-left associativity, so $a /= b /= c$ is actually `a /= (b /= c)`; so if $a = 4$, $b = 2$, $c = 2$, then the result is 4.
- Every expression will finally construct a tree (not definitely "binary", e.g. `?:` is triplet, `++` is unary).

Expression

- Then, it's order of expression evaluation that computes the whole tree.
 - It is only determined that before the evaluation of root, the left child and the right child will be evaluated first; the order is **unspecified**.
 - e.g. $f1() +_1 f2() +_2 f3()$, it's $root(+_2) \rightarrow lChild(f1() + f2()) \rightarrow rChild(f3())$, while $lChild$ is $root(+_1) \rightarrow lChild(f1()) \rightarrow rChild(f2())$;
 - We can know before $+_1$ is evaluated, $lChild$ and $rChild$ is first evaluated.
 - However, you can evaluate in the sequence of:
 - $lChild$ evaluates $f1()$
 - $rChild$ evaluates $f3()$, gets the value.
 - $lChild$ evaluates $f2()$, gets the value.
 - This still obeys our rules, e.g. $f1()$ and $f2()$ evaluated before $lChild$.
 - So if we output a in $f1()$, b in $f2()$, c in $f3()$, any permutation of abc is possible!
 - To sum up, evaluation order is hugely determined by how compiler computes the tree.

Expression

- Besides, some operators have side effects, e.g. `++`, `/=`.
 - So it's more complex; when will the side effect happen?
 - E.g. infamous Tan-C, `++num + num`, where `num = 1`
 - It's determined that `++num` is 2, but if the side effect of `++` happens before the second evaluation, then the second `num` is 2; otherwise it's 1. So, it's possibly 3 or 4!
 - It's **undefined** (i.e. UB) rather than unspecified, meaning that the same compiler may behave differently.
- There are some rules:
 - For `&&` and `||`, since they have short-circuit property, the first part will be fully evaluated, i.e. value computation and side effects will happen before the second part.
 - For a function call, all parameters (including `a` for e.g. `a.Func()` or `a->Func()`) are fully evaluated before entering the function.
 - For `condition ? exp1 : exp2`, `condition` is fully evaluated before `exp1` and `exp2`.
 - For comma(`,`), sub-expressions are fully evaluated one by one.

Expression

- Since C++17, many new rules are added so that it's more intuitive to humans.
 - Parameters in function are evaluated indeterminately, i.e. every sub-tree represented by the parameter is fully evaluated in a **non-overlap** way!
 - This is important in some cases, we'll cover it later.
 - Every overloaded operator has the same evaluation rule as the built-in operators, rather than only be treated as a function call.
 - For $E1[E2]$, $E1.E2$, $E1 \ll E2$, $E1 \gg E2$, $E1$ is always fully evaluated before $E2$.
 - For $E1 = E2$, $E1 @= E2$, $E2$ is always fully evaluated before $E1$.
- With these rules, some weird UB expressions before become valid, e.g. $num = num++ + 2$ where $num = 1$ will always get 4.
 - Before, it may be 2 since the side effect of $num++$ may happen after $=$, so it's UB.

Expression

- However, things like `n = ++num + num++` is still UB.
- All in all, it's discouraged to compound lots of expressions with side effects, both in the view of UB or debugging.
 - Since they all happen in a single step, so when a bug lies there, you cannot step into to find it.
 - By decomposing them to separate expressions, it's more human-friendly.
- There are also other important things for expression; we'll cover them in *Move Semantics*.

Expression

- More useful examples: chained call

- Even if you're not familiar with `std::string`, you can guess that it wants to generate "it sometimes works if I believe".

```
std::string s = "I heard it even works if you don't believe";  
s.replace(0,8,"").replace(s.find("even"),4,"sometimes")  
  .replace(s.find("you don't"),9,"I");
```

- This only holds water if parameters of the first `.replace` are evaluated, then it's called, then parameters of the second `.replace` are evaluated, etc.
 - As a normal man will think so!
- However, weird things are that all `s.find` may be all evaluated before any call of `.replace`, because before C++17 it's not guaranteed that things after `.` is evaluated after things before `..`
 - Then the index doesn't mean something in the new string, but all in the original string! It may be even out of range then.

Class

- We finally step into 程设... (You may have a small rest if you feel tired 😊)
- Admittedly, this course makes you know basic C++ in a very C++98 way (not criticize).
 - We'll correct many of them, kick off the obsolete and embrace new and effective ways.
- Class Type
 - Member Functions & Data Members
 - Ctor and Dtor
 - Access Control
 - Inheritance

Basics

Class

Basics

- Class
 - Ctor and Dtor
 - Member Functions
 - Access Control
 - Inheritance

Ctor&Dtor

- Class extracts features in common from a kind of thing in the real life, **encapsulates** in the program by data (i.e. data members) and methods (i.e. member functions) to save states and send messages.
- In OOP, we usually call the instances of a class **object**.
 - Constructor is used to initialize data members of objects.
 - A kind of special ctor is copy ctor, which accepts an object of the same class and copy(& other necessary ops) data members from them.
 - Destructor is used to destroy them.
 - You may also define functions in class.

Ctor&Dtor

- Previously, what we learn is...

```
class Person
{
public:
    Person(const std::string& initName) {
        std::cout << "Construct!\n";
        name_ = initName;
    }
};
```

```
~Person(){
    std::cout << "Destruct!\n";
}

private:
    std::string name_;
};
```

- However, in fact `name_` has been default-initialized before ctor enters the function body; what we call is actually `operator=`.
 - If some objects can only be created but not assigned (e.g. `const` members, reference members), this will be wrong.
 - It's also less efficient if the default initialization is costly.

Notice that once the reference member is initialized, it cannot change the referred object. So if the object is actually destroyed, then the member is dangling!

Ctor&Dtor

- The recommended way is *member initializer list*:

- `member1{...}, member2{...}, ... { /* function body */ }.`

```
Person(const std::string& initName) : name_{initName} {  
    std::cout << "Construct!\n";  
}
```

- `{}` is used since C++11; in C++98, it should be replaced with `()`; we'll tell their differences sooner.
- If you declare ctor explicitly, the default one is disabled.
 - If the default ctor is just `Class(){} (i.e. do nothing except for default initializing all members)`, you'd better use `Class() = default`;
 - `=default` has more usage and we'll cover it later.
 - You can also use `=delete` to make it not callable (e.g. disable copy ctor).
 - Any function can be explicitly disabled by `=delete`!

This ctor can be improved; we'll cover it in *Move Semantics*.

```
bool isLucky(int number);           //原始版本  
bool isLucky(char) = delete;        //拒绝char  
bool isLucky(bool) = delete;        //拒绝bool  
bool isLucky(double) = delete;      //拒绝float和double
```


Ctor&Dtor

- If the member has the default value, you may use *In-Class Member Initializer* since C++11 (**recommended**) rather than `name_{"Unknown"}` in ctor.
 - Then to sum up, it is:
- If you don't provide one, `std::string` has its own default initializer (usually make it empty) and it'll be called. Only types that don't have default ctor (e.g. `int`) will be random.
- In dtor, all members will call dtor automatically; `std::string` has proper dtor, so dtor does nothing here.
 - However, if we e.g. use a pointer `newed` in ctor, we need to `delete` it manually in dtor because destruction of pointer doesn't release the memory.
 - RAI: dtor clears all resources required in ctor.



```
class Person
{
public:
    Person() = default;
    Person(const std::string& initName) : name_{initName} {
        std::cout << "Construct!\n";
    }

    ~Person(){
        std::cout << "Destruct!\n";
    }

private:
    std::string name_ = "Unknown";
};
```

Ctor&Dtor

- So what's the order of constructing & destructing data members?
 - Ctor initializes them in the same order of their declaration; destruction happens reversely (like a stack).

```
class A { public: A() { std::cout << "CA.\n"; } ~A() { std::cout << "DA.\n"; } };  
class B { public: B() { std::cout << "CB.\n"; } ~B() { std::cout << "DB.\n"; } };  
class C
```

- **IMPORTANT:** It's dangerous if you use members behind to initialize previous members in ctor!
 - So, it's recommended to initialize members in list initializer in the same order of declaration
 - Some compilers will report warnings if you disobey it.

```
A a;  
B b;  
};  
  
int main()  
{  
    C c;  
    return 0;  
}
```

CA.
CB.
DB.
DA.
请按

Initialization of Object

- Before, we've learnt how to define object of some class.
 - E.g. `A a(1);` or `A a = 1;`.
- Since C++11, *Uniform Initialization* is introduced.
 - (Almost) all initialization can be done by curly bracket `{}`.
 - Compared to `()`, it will strictly check *Narrowing Conversion*;
 - i.e. the converted type cannot represent all values.
 - e.g. if an `uint32_t` is used to initialize `uint16_t`, then compiler reports an error; you need to convert it manually if you want.
 - Even `uint32_t` to `int32_t` and vice versa forbidden, since the latter has negative values which cannot be represented by the former, and the former has larger positive values which cannot be rep.ed by the latter.
 - But if this value can be determined in compile time and it's representable, then it's still valid (e.g. `const uint32_t a = 2 => uint16_t`).
 - This facilitates type safety.
 - It can also prevent *most vexing parse*, e.g. `Class a();/Class a(C2());` will be seen as a function prototype instead of a variable!

Initialization of Object

- There are many ways for an variable to be initialized in C++.
- Default initialization: `T a, new T;`
 - If `T` has a default ctor, it will be called.
 - Otherwise the memory is random, e.g. if `T = int`, then it's possibly anything. It's **UB** to read them before assigning them.
 - Members like this are also random if you forget to initialize them in ctor.
- Value initialization: No parameter for initialization; `T a(), T a{}, new T{}, new T()`.
 - Compared to default initialization, this will *zero-initialize* other values, e.g. `int a{};` will make `a == 0` while `int a;` just makes it random.
 - It's generally (not definitely, we may cover it in the future) recommended to use value initialization over default initialization.

Initialization of Object

- Direct initialization: what we learn before, `T a(xx, yy, ...)`, `T(xx, yy, ...)`, `new T(xx, yy, ...)`, `T a{xx}` for non-class type `T`.
 - BTW, `T(xx)` and direct list initialization (covered later!) `T{xx}` are also seen as explicit conversion, which is effectively same as `(T)xx`.
- Copy initialization: `T a = xx`; `T a[] = {xx,...}`; pass normal param & return normal val.
 - Ctors that use `explicit` cannot use this way.
 - Before C++17, this also requires available copy ctor.
 - This is because *copy elision*; we'll cover it in the future!

```
class A { public: explicit A(int a) {} };  
void Func(A a) {}  
int main()  
{  
    A a = 1;      // error  
    A a = A(1);   // right  
    Func(1);      // error  
    Func(A(1));   // right  
    return 0;  
}
```

Initialization of Object

- The following two ways are introduced since C++11; they hugely unify initialization.
 - You may not recognize them, just remember you may just add {} to initialize anything!
 - No type is needed!

```
Sphere(Vector3 init_position, Vector3 init_emission, double init_radius, Vector3 init_color, ReflType init_type, double init_n = 1)
scene.AddObject(new Sphere{ {99 - 1e5, 40.8, 81.6}, {0, 0, 0}, 1e5, {192, 64, 64}, Object::ReflType::DIFFUSE });
std::tuple<int, double, bool> SearchIntersect(const Ray& ray) const
{
    return { finalID, finalDistance, finalInside};
}
```

- There is only one exception, and we'll cover it in *Container*.

Initialization of Object*

- List initialization:
 - Direct list initialization: `T a{xx, yy, ...}, T{xx, yy, ...}, new T{xx,yy,...}.`
 - Copy list initialization: `T a={xx,yy, ...}, func({xx, yy, ...}), return {xx, yy, ...}, [{xx, yy, ...}]` (e.g. for index of `std::map`)
 - Still need non-explicit.
 - Parameters of list initialization are evaluated indeterminately, just like a function call.
- Aggregate initialization: a special form of list initialization.
 - “Aggregate” means array and some classes (we’ll give you precise definition sooner); you can initialize member by member without a ctor!
 - `int a[3] = {1,2,3}` or `int a[3]{1,2,3}.`
 - `A a={1,2,3}` or `A a{1,2,3}` for e.g. `class A{public: int a; int b[2] };`
 - Containers like `std::vector` can also use `{1,2,3}` so that it’s consistent with normal array(**Uniform**)!
 - But this is implemented through *initializer_list*, we’ll cover in *Container*.

Initialization of Object

- Notice that `auto` will behave in a weird way...
- `auto a = {1}`, `auto` is initializer list!
- `auto a{1}`, `auto` is `int` since C++14 (C++11 is still initializer list).
- This is one of the most confusing facts for novices in C++, just remember it now!
- Personally, I recommend not mixing up `auto` and `{}`;
 - When I use `{}`, I will always specify type explicitly.
 - E.g. `A a{1}`;
 - When I use `auto`, there will be no things like `= {xx}`, but only `= xx` or `= Type{xx}`.
 - E.g. `auto a = 1`; `auto a = A{1}`; `auto a = Func({xx, yy}, zz)`;

Ctor&Dtor

Disable copy ctor/assignment:

```
Person(const Person&) = delete;  
Person& operator=(const Person&) = delete;
```

- Copy Ctor:

```
public:  
    Person(const Person& another) : name_{ another.name_ } {}
```

- `Person p1 = p2;` will call it!

- It's used for construction; only `p1 = p2` will call `operator=`.
- This is actually operator overloading.

- Pay attention to self-assignment: if you hold e.g. a pointer, and you may `delete` the original one, `new` the new one, and copy the content from `another`, then judge `this != &another!`

```
Person& operator=(const Person& another) {  
    name_ = another.name_;  
    return *this;  
}
```

- Otherwise, the original resource is released before copying.
- For classes in standard library, it's safe to self-assign.
- We'll improve this part and introduce some general rules for ctor & `operator=` in *Move Semantics*.

Member Functions

- All non-static member functions in a class implicitly have a `this` pointer (with the type `Class*`) as parameter.
 - All members are accessed by `this` implicitly.
- Sometimes you may hope methods unable to modify data members, then you need a `const`:
 - This is achieved by make `this` to be `const Class*`.
- But, what if what we change is just status that user cannot know?
 - E.g. We need a mutex as member to protect reading, and it's totally transparent to the user. However, in `const` method, we cannot acquire/release the mutex since it will modify the mutex.
 - C++ introduces a keyword `mutable`.
 - E.g. you may declare `mutable Mutex myLock;` in the class.
 - You may modify `mutable` variable in `const` method.
 - Use `mutable` carefully in restricted cases.

Static Member Functions

- For static methods, they can be called both by `Class::` or `obj.`; it's just like a normal function with `Class::`.
 - You can take its address normally.
 - Since it doesn't bind on an object (i.e. no `this`), it cannot specify `const`.

```
class Vector3
{
public:
    Vector3(float x0, float y0, float z0) : x{ x0 }, y{ y0 }, z{ z0 }{}
    static Vector3 Zero() { return { 0.f, 0.f, 0.f }; };
private:
    float x, y, z;
};

int main()
{
    Vector3 vec = Vector3::Zero(), vec2 = vec.Zero();
    return 0;
}
```

Basics

- Class
 - Ctor and Dtor
 - Member Functions
 - Access Control
 - Inheritance

Access Control

- In strict OOP, all data members should be hidden from users, and objects communicate with each other only by messages.
 - To achieve the “hidden” property, **access control specifiers** are needed.
 - In fact, most of languages that support OOP don't strictly require the user to hide all data; but it's recommended to do so.
- `private/protected/public; class` is `private` by default when no specifiers are provided.
- `protected` is used in inheritance, which will be covered later.

Access Control

- Sometimes we may want to hide data members totally from users (i.e. even no way to get it), while we still want to expose them to some class (e.g. some helpers).
 - We need to specify a way to loosen the restriction between classes – that is **friend**.
 - If class A declares class B as its friend, then B can access all members of A.
 - If A says B is its friend, B can take anything from A.
 - Friend only exists in the exact two classes; inheritance doesn't affect it; no transitivity among friends.
 - A class can also declare a method as **friend**.

Access Control

- You may also declare only part of methods rather than all methods in a class as friend functions.
 - It's more complex, and we'll cover it in *Programming in Multiple Files*.
- Finally, you can also define a class in a class, i.e. *nested class*, which will be affected by access control specifiers too.

```
class A {  
    int a;  
public:  
    class B {};  
};  
A::B b;
```

```
class A  
{  
    friend class B;  
    friend void test(A& a);  
public:  
private:  
    int val;  
};  
  
class B  
{  
public:  
    B(A& a) { a.val; A a2; a2.val; }  
};  
  
void test(A& a) { a.val; A a2; a2.val; }
```

Basics

- Class
 - Ctor and Dtor
 - Member Functions
 - Access Control
 - Inheritance

Inheritance

- Sometimes, one class is a subdivision of another class, e.g. **Student is a Person**, but has more properties than **Person**.
 - We say **Student** as **child class** or **derived class**, while **Person** as **parent class** or **base class**.
 - **Student** can use all **public** and **protected** members in **Person**, and it can extend its own methods.
 - Inheritance can reduce code duplication.
- Through **polymorphism**, you can use the base class to load the derived object and call its own methods.
- However, inheritance has many possible problems and is even dropped out in some new languages. It's preferred to use **composition** instead of inheritance if you can.

Inheritance

- public inheritance is the most common.
- Call initializer of the base class in this way.
- **protected** can be accessed.
 - **private** cannot.
- Derived class can be implicitly converted to base class.
- The parent is fully constructed before constructing members of the child.
 - The parent is destructed after call dtor of the child, just like stack.

```
class Person
{
public:
    Person(const std::string& initName, const std::string& initID):
        name{ initName }, idInIDCard_{ initID } {}
    std::string name;
protected:
    std::string idInIDCard_;
    void InputIDCard_(){}
private:
    int secret = 0;
};

class Student : public Person
{
public:
    Student(const std::string& initName, const std::string& initID,
            const std::string& initSchool) :
        Person{ initName, initID }, school{ initSchool }
    {
        idInIDCard_; // right;
        InputIDCard_(); // right;
        secret; // private cannot be accessed.
    }
    std::string school;
};
```

Polymorphism

- However, in C++, polymorphism is valid **only in pointer and reference.**

```
class Person
{
public:
    Person(const std::string& initName): name_{ initName } {}

    virtual void PrintInfo() const {
        std::cout << std::format("This is {}.\\n", name_);
    }
private:
    std::string name_;
};
```

```
class Student : public Person
{
public:
    Student(const std::string& initName, const std::string& initSchool) :
        Person{ initName }, school_{ initSchool } {}

    virtual void PrintInfo() const {
        Person::PrintInfo();
        std::cout << std::format(" I'm from {}.\\n", school_);
    }
private:
    std::string school_;
};
```

```
void PrintInfo(const Person& person) {
    person.PrintInfo();
}
```

```
int main()
{
    Person p{ "Tom" };
    PrintInfo(p);

    Student s{ "Jerry", "T&J"};
    PrintInfo(s);
    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

```
This is Tom.
This is Jerry.
  I'm from T&J.
```

Polymorphism

- We know that typical implementation of polymorphism in C++ is through **virtual pointer** and **virtual table**.
 - Every object whose class has a virtual method will have a virtual pointer, which points to virtual table of its class.
 - E.g. `Person` will have a pointer to `Person`'s virtual table, while `Student` will have a pointer to `Student`'s.
 - Then, when the pointer of `Person` call `PrintInfo`, it will use the virtual pointer to jump to the virtual table and finds the real function.
 - So even though the underlying `Person` is `Student`, `PrintInfo` is still `Student`'s method.
 - Slightly slower than normal functions since an additional jump is needed.
 - Compilers may have slightly different implementation, but roughly like this.

Polymorphism

- In C++11, it's recommended to use **override**.

```
void PrintInfo() const override {  
    Person::PrintInfo();  
}
```

- When you doesn't override a virtual method in the parent class, the compiler will report an error.
 - This can e.g. prevent you from leaving out **const**, **unsigned**, etc..
 - In C++98, you may just create a new method, and only know your mistake when you find that there is no polymorphism in **runtime**.
- There is another similar specifier **final**.
 - It means override, and the derived class cannot override again.
- E.g. For **Student**:
 - Then we add class **S2**:

```
void PrintInfo() const final {  
    Person::PrintInfo();  
}
```

```
class S2 : public Student  
{  
public:  
    void PrintInfo() const override {};  
};  
void S2::PrintInfo() const
```

联机搜索

无法重写"final"函数 "Student::PrintInfo" (已声明 所在行数:23)

Polymorphism

- You can also make a class unable to be derived, e.g. `class Student final {...};`
 - This is called sealed class in many languages, e.g. C#.
- As its name, you hope it's the final node in the override chain.
 - Final entities can be more efficient than normal virtual methods; it may do optimization in virtual table.
- That is, when compiler can determine the actual type of `Person`, e.g. it knows that it's definitely `Person` or definitely `Student`, then an optimization called **devirtualization** will happen.
 - i.e. the function call happens directly, not through the virtual pointer.
 - So, for the simplest case, a `final` class, e.g. `S2`, can be devirtualized since `S2` can only be `S2`.

There are also many other analysis to devirtualize; you may check [GCC对C++虚函数调用的一个优化 - RednaxelaFX的文章](#) and [When can the C++ compiler devirtualize a call? - syheliel的文章](#) for more details.

Polymorphism

- If a class just specifies things that derived class **should** have, but doesn't specify implementation itself, then it's called **abstract class** (ABC).
 - So, abstract class cannot be instantiated, i.e. have object.
 - However, you can use its pointer, so that it may refer to derived objects.
- C++ implements abstract class by **pure virtual** function.
 - They have only declaration and cannot have definition.
- Derived classes that want to generate instances should implement **PrintInfo**; otherwise it's still abstract class.
- It's UB to call pure virtual function by vptr.

```
class Person
{
public:
    virtual void PrintInfo() const = 0;
private:
    std::string name_;
};
```

Polymorphism

- Calling pure virtual function seems to be banned by compiler, so how can you do that?
 - Only when an object is completely created can **virtual** makes polymorphism; otherwise it's UB.
 - Or you may directly think vptr is initialized **after** ctor exits the function body.
 - So, in the ctor of the derived class, the vptr actually points to the vtable of the base class.
 - When you call a virtual function in a normal function, **this** actually participates in so that it will call the function according to vptr.
 - Making use of this, you may skip the check of the compiler because compiler doesn't know which virtual method the normal function calls!
 - Then, if you call this normal method in ctor, and this normal method calls a pure virtual method, then compiler doesn't find it but an runtime error happens (e.g. msvc reports runtime error "pure virtual function call").

Polymorphism

```
class Base
{
public:
    void reallyDoIt() { doIt(); }
    virtual void doIt() { std::cout << "Base!\n"; };
};

class Derived : public Base
{
    void doIt() override { std::cout << "Derived!\n"; }
};

int main()
{
    Derived d;

    Base& b = d;
    b.reallyDoIt();
    return 0;
}
```



A terminal window showing the execution of the program. The title bar reads "C:\WINDOWS\system32\cmd.exe". The output is "Derived!" followed by a prompt "请按任意键继续. . .".

```
class Base
{
public:
    Base() { reallyDoIt(); }
    void reallyDoIt() { doIt(); }
    virtual void doIt() = 0;
};

class Derived : public Base
{
    void doIt() {}
};

int main(void)
{
    Derived d; // error!
    return 0;
}
```

Polymorphism

- Besides, since members of children have been destructed when calling dtor in parent, it's dangerous too...
- To conclude, don't call **any** virtual function and any function that calls virtual function in ctor & dtor!
- Besides, you should usually make dtor of base class **virtual**.
 - Reason: deleting Base* that cast from Derived* will lead to correct dtor.
- Ctor cannot be pure virtual but dtor can.

Some Covert Facts in Inheritance

- Note0: The meaning of “override” is not specialized for virtual methods like the keyword **override**.
 - If you name a non-virtual function with the same name as parent’s, you’ll also override it.
 - You need to use **Parent::Func()** to call the parent version.
- Note1: **private** inheritance usually denotes the relation of **has-a**.
 - You cannot use C++-style cast to convert **Derived*/&** to **Base*/&** in **private** inheritance.
 - But that can be replaced by composition, which is also better than inheritance.
 - It may be used in legacy code...

Some Covert Facts in Inheritance

- Note2: C++ allows limited shift of return type in virtual override.
 - We say that overriding virtual functions requires same parameters and return type.
 - But, you can return pointer/reference of child object if the overridden method returns pointer/reference of parent object.
 - Obviously, you can use `Base*` to both accept `Base*` and `Derived*`.
 - For example (we extract them from the definition of class):
 - Notice that smart pointers that will be covered in the future cannot be returned, since `SmartPtr<Base>` is not the parent of `SmartPtr<Derived>`.

```
Cherry* CherryTree::Pick() { return new Cherry{}; }
BingCherry* BingCherryTree::Pick() { // 在声明中可以写override
    std::cout << "Bing!\n";
    return new BingCherry{};
}

CherryTree* tree = new BingCherryTree{};
Cherry* cherry = tree->Pick();
```

Some Covert Facts in Inheritance

- Note3: When virtual methods have default params, calling methods of **Derived*/&** by **Base*/&** will fill in the default param of **Base** methods!

- E.g.

```
void Parent::Go(int i = 2) { std::cout << "Base's go with i=" << i <<
"\n"; };
void Child::Go(int i = 4) { std::cout << "Derived's go with i=" << i <<
"\n"; }

Child child;
child.Go(); // Derived's go with i=4
Parent& childRef = child;
childRef.Go(); // Derived's go with i=2
```

- This is because default params are filled in by type in the compilation time, while virtual methods are called through jump in runtime! So the compiler can only use **Base's** (unless there are something like virtual table for default params, but that's costly).
- Usually you should make default params of virtual methods consistent...
 - E.g. give a name to the default values.

Some Covert Facts in Inheritance

- Note4: There is a pattern called *template method pattern* that utilize **private** virtual method.
 - The base class is called *template class*.
 - It'll call virtual methods in its methods so that the polymorphism can be utilized.
 - What derived classes need to do is just overriding those virtual methods, so the template class will automatically adjust them to intension of the derived.
- Note5: You can change the access control specifier of virtual methods, but it'll collapse the access barrier, so it should be avoided to use.

```
class Student
{
public:
    float GetGPA() { return GetGPACoeff() * 4.0;}
private:
    virtual float GetGPACoeff(){ return 0.8; };
};

class JuanWang: public Student
{
    virtual float GetGPACoeff(){ return 1.0; };
};
```

Some Covert Facts in Inheritance

- Note6: Pure virtual functions can have definition, but it should be split from the prototype in the class.
 - However, this definition cannot be called by vptr, i.e. here it's still UB!
 - You can only call it by `Base::DoIt()` explicitly.
 - It has several usage:
 - 1. if you define a pure virtual dtor, it **must** have a definition (though likely do nothing).
 - That's because dtor of derived class will always call base dtor.
 - A pure virtual dtor is usually used to force an ABC.
 - 2. if you hope to provide a default behavior, but don't want the derived class to directly accept it.
 - If you use non-pure-virtual one, then the derived can choose to not define it again.
 - But for this one, you need to define it and call `Base::` explicitly; this is to prevent you from forgetting to re-define it when you add a new derived.

```
class Base
{
public:
    Base() { ReallyDoIt(); }
    void ReallyDoIt() { DoIt(); }
    virtual void DoIt() = 0;
};

void Base::DoIt(){}
```

Final word: struct

- `struct` is almost same as `class` in C++, except that `struct` use `public` as the default access control.
- However, `struct` is usually used when only data members exist and are **all public**.
 - They shouldn't have member functions. At most it has ctor & dtor & some operators.
 - In C#, this is regulated in the language, with some other restrictions.
- With these constraints, `struct` will be an aggregate, which can use aggregate initialization.
 - Remember? E.g. `a{1,2,3}` without explicit ctor!
 - Since C++20, aggregate can also use **designated initialization**!

Aggregate in C++20*

See [here](#) for more details about what an aggregate is in C++ evolution.

To summarize, since C++20, an *aggregate* is defined as follows:

- Either an array
- Or a *class type* (class, struct, or union) with:
 - No *user-declared* constructor
 - No constructor inherited by a using declaration
 - No private or protected non-static data members
 - No virtual functions
 - No virtual, private, or protected base class

To allow you to *initialize* aggregates, the following additional constraints apply:

- No private or protected base class members
- No private or protected constructors

Credit: C++20 – The Complete Guide By *Nicolai M. Josuttis*. I used to recommend PKU library to purchase this book and it was accepted; you can borrow it if you would like!

Designated Initialization

- Data members should be designated in the same order of declaration.
- This is human-friendly.
- C has this feature (and more powerful) since C99.
 - E.g. the order can be random.

```
struct Priority { int val; };  
  
struct Point  
{  
    int x, y;  
    Priority priority[2];  
};  
  
int main()  
{  
    Point p{ .x = 1, .y = 2, .priority = { {.val = 1}, {.val = 2} } };  
    return 0;  
}
```

- Final word: array cannot use designated initialization though it's an aggregate.

Bit field*

- There exists a special kind of “**struct**” called bit field.
 - This is usually used in C previously to save memory.
 - If bit amount exceeds the type(e.g. use 9 bits for **unsigned char**), it will be truncate to the max bits.

```
struct S
{
    unsigned char s1 : 3; // 3 bits.
    unsigned char   : 2; // unused 2 bits.
    unsigned char s2 : 6; // only remain 3 bits in 1st byte
                        //      -> start at 2nd.
    unsigned char s3 : 2; // 2 bits.
    unsigned char s4 : 3; // 3 bits, in 3rd byte.
    unsigned char   : 0; // 0 means start from a byte
    unsigned char s5 : 2; // the fourth byte.
};

const int s = sizeof(S);
```

const int s = 4

联机搜索

Bit field*

- Much syntax in normal `struct` in C++ is invalid in bit field, and you may treat it as C-like `struct`.
 - For example, you cannot pass it by reference in function arguments.
 - Click [here](#) for more info.
- You may use bit field as flag collection, e.g. enable dark mode, enable mute, etc., with each one occupying 1 bit.
- This syntax is possibly not that useful now, since memory is not that tight in modern computers, and it may hurt performance since it needs to operate on bits rather than bytes, which needs extra efforts.

Basics

Function Overloading

Function Overloading

- In C++, functions can have the same name with different parameters.
 - This is prohibited in C.
- This is done by compilers using a technique called **name mangling**.
 - Though two functions seem to have the same name, the compiler decorates each one to a unique symbol from the parameters and their types.
 - Return type does **NOT** participate in name mangling!
 - Except that it's a template parameter, since all template parameters are part of mangled name.
 - E.g. for function `namespace Namespace {int function(int x);}`, it's mangled in msvc as `?function@NameSpace@@YAHH@Z`, while in gcc as `_ZN9NameSpace8functionEi`.
You may demangle them [in this website](#).

Operator Overloading

- A special overloading is operator overloading.
- For an object of some class, operators may have different meanings by providing overloaded operators.
 - The operator will still accept the same operands, and you cannot define new symbol as operator.
 - But the type of operands are not required to be same
e.g. you may write `operator+(int, Vector3)`
- You may write them as class methods:
 - In fact `a.operator+(b)`, so it's obligation to make the object the first operand.
- Or as global functions:
 - In fact `operator+(a, b)`, so `a` may be implicitly converted to `Vector3` when `b` is `Vector3`.

```
Vector3 a{ 1,2,3 }, b{ 4,5,6 };  
Vector3 c = a + b;
```

```
Vector3 operator+(const Vector3& v) const {  
    return Vector3{ x + v.x, y + v.y, z + v.z };  
}
```

```
friend Vector3 operator+(const Vector3& v1, const Vector3& v2);  
Vector3 operator+(const Vector3& v1, const Vector3& v2) {  
    return Vector3{ v1.x + v2.x, v1.y + v2.y, v1.z + v2.z };  
}
```

Or you may use `GetX()`, etc. if `friend` is not available.

Operator Overloading

- You can overload:

- `+, -, *, /, %, |, &, ^, <<, >>`: recommend global functions; return new object.
 - Specially, `<<` and `>>` can be used as insertion and extraction operator, which must be global functions since you cannot add methods for the stream class; we'll show you sooner.
- `+=, -=, *=, /=, |=, &=, ^=, <<=, >>=`: require member functions, since the first operand **must** be a "named" object; return reference(i.e. `*this`).
- `Prefix++&--`: unary, require member function, return `*this`.
- `+, -, ~, postfix++&--`: unary, recommend member function, return new object.
 - Specially, postfix `++/--` have an unused parameter `int`, which is used to distinguish the prefix and postfix.

Operator Overloading

- `*`, `->`: usually used in e.g. some wrapper of pointers; `const` is needed if the pointed element is `const`.
 - Particularly, `->` is actually `(a.operator->())->`.

```
int main()
{
    MyAPtr aPtr{ 0 };
    A& a = *aPtr;
    int val = aPtr->a;
    return 0;
}
```

```
struct A { int a; };

class MyAPtr
{
public:
    MyAPtr(int a0) : ptr{ new A{a0} } {}
    ~MyAPtr() { delete ptr; }

    A& operator*() { return *ptr; }
    const A& operator*() const { return *ptr; }
    A* operator->() { return ptr; }
    const A* operator->() const { return ptr; }

private:
    A* ptr;
};
```

Operator Overloading

- `&&`, `||`: rarely overloaded, since short-circuit property of evaluation in Boolean expression doesn't work.
 - E.g. `p && p->flag` makes use of short circuit so that `p->flag` will not dereference `nullptr`; However in `operator||(a, b)`, `a` and `b` will be evaluated before the function is called, so short-circuit disappears.
 - However, Catch2 uses it; we'll cover it in *Error Handling*.
- `!`: rarely because usually conversion to `bool/void*` is provided so that `!` will be explained as negation of that `bool/void*`.
- `new`, `new[]`, `delete`, `delete[]`: we'll cover them in *Memory Management*.

Operator Overloading

- `,`: Yes, you may even overload comma, but even more rarely.
 - However, Boost.Parameter, Boost.Assign and Eigen have utilized this. They're all very basic libraries.
 - You may check [here](#) for comma overloading usage.
- `&`: Get the address; incredibly rare.
 - So if you want to write some basic library, you should use `std::addressof()` defined in `<memory>` to get the exact address.
- You cannot overload `.`, `?::`, `::`, `.*`.
- The usage of `.*` and `->*` may be covered in the future.

Operator Overloading

- `<<` and `>>` in stream:
- You can write them in class, though they are in fact global functions.
 - Compilers can find them by **ADL** (Argument-Dependent Lookup).
 - Briefly, compiler will also find methods in the scope of params, so `std::cin >> a` will also find in `Real`.
 - **Name lookup** is actually more complex, but we don't cover it in this course.
 - We'll also explain why you need to overload them like this in the future.

You may go [here](#) if you're interested in name lookup.



```
#include <iostream>

class Real
{
public:
    friend std::istream& operator>>(std::istream& is, Real& r) {
        is >> r.val;
        return is;
    }

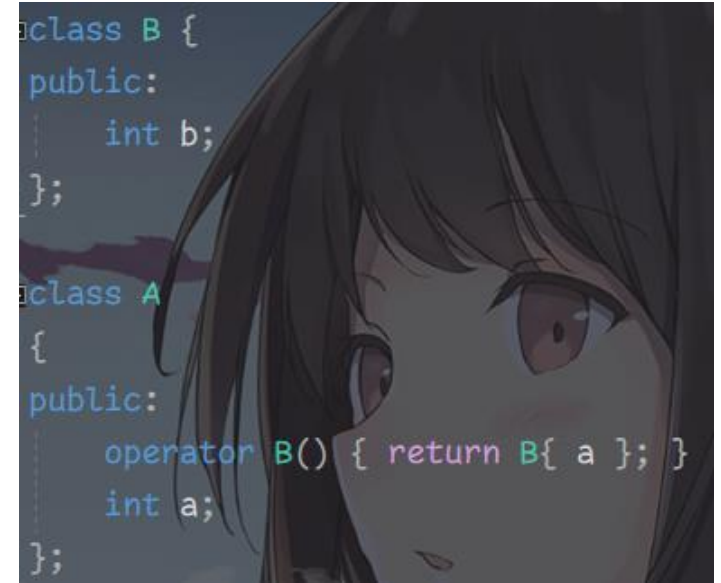
    friend std::ostream& operator<<(std::ostream& os, const Real& r) {
        os << r.val;
        return os;
    }

private:
    float val;
};

int main()
{
    Real a;
    std::cin >> a;
    std::cout << a << '\n';
    return 0;
}
```

Operator Overloading

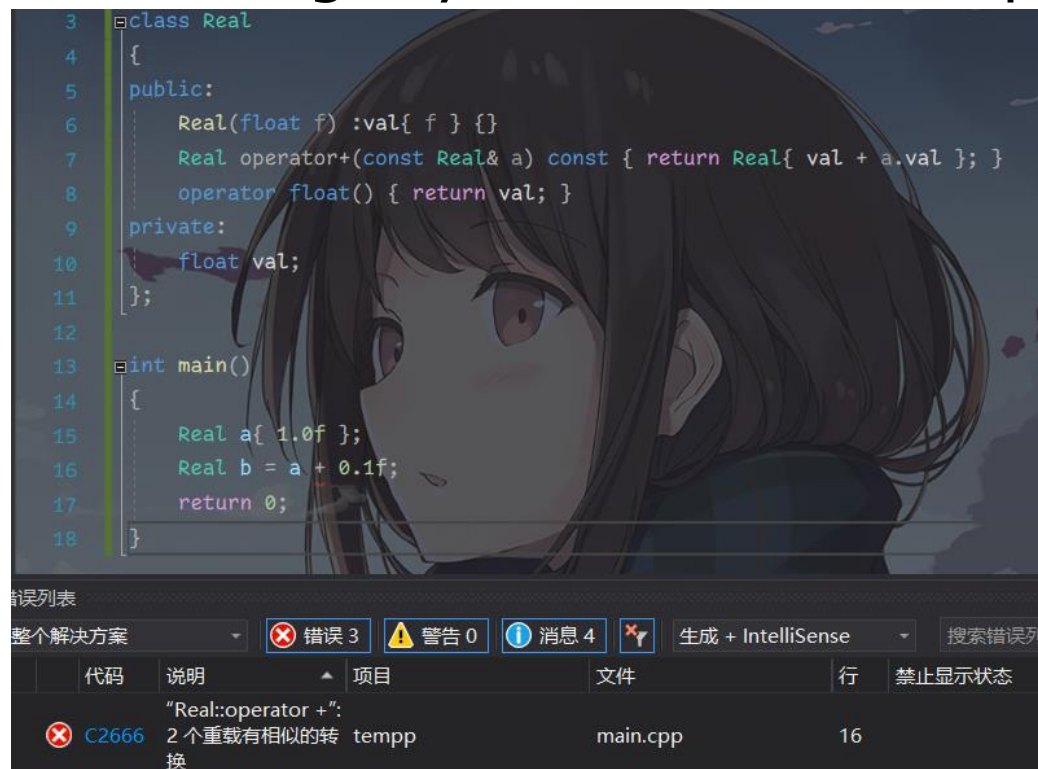
- We'll introduce some special operators then.
- Conversion Operator:
 - No return type needed since the operator name just specifies it.
 - Notice that this is similar to add a ctor for B.
 - Similarly, you can add **explicit** to force user to convert it manually.
- Particularly, for pointer class converted to **bool**, maybe to **void*** is more proper so **p != nullptr** is valid.
- You can use **operator auto(){ ... }** so that the type will be deduced from the return value.



```
class B {  
public:  
    int b;  
};  
  
class A  
{  
public:  
    operator B() { return B{ a }; }  
    int a;  
};
```

Operator Overloading

- Conversion operator may cause ambiguity so that the compiler reports error.
- $a + 0.1f$ is possibly:
 - `float(a) + 0.1f`,
i.e. adding two `float`.
 - `a + Real{0.1f}`,
i.e. adding two `Real`.
- Though they're same in result, the compiler doesn't know that.
- You may delicately write ctor and conversion operator to prevent them from conflict (or just use `explicit` to disable implicit conversion).



```
3 class Real
4 {
5 public:
6     Real(float f) : val{ f } {}
7     Real operator+(const Real& a) const { return Real{ val + a.val }; }
8     operator float() { return val; }
9 private:
10    float val;
11 };
12
13 int main()
14 {
15     Real a{ 1.0f };
16     Real b = a + 0.1f;
17     return 0;
18 }
```

错误列表

整个解决方案 错误 3 警告 0 消息 4 生成 + IntelliSense 搜索错误列表

代码	说明	项目	文件	行	禁止显示状态
C2666	"Real::operator +": 2个重载有相似的转换	temp	main.cpp	16	

Operator Overloading

- `<, >, <=, >=, ==, !=`: You may find it annoying to overload them all with many duplicate codes...
 - `a < b` is right, but `a >= b` compiles error???
 - The standard library mostly uses `<` and `==` to get all relations, so that the burden can be reduced.
 - But this is still confusing for users.
- In C++20, this problem is completely solved by spaceship operator/ three-way comparison operator `<=>`.
 - First, let's see how you get benefit from it!
 - `<=>` will deduce the first four operators.
 - `==` will deduce `!=`.

```
#include <compare>
struct DijkstraInfo
{
    int vertexID;
    int currDis;
    auto operator<=>(const DijkstraInfo& another) const {
        return currDis <=> another.currDis;
    }
    bool operator==(const DijkstraInfo& another) const {
        return currDis == another.currDis;
    }
};

int main()
{
    DijkstraInfo a{ 0, 1 }, b{ 1, 2 };
    a >= b, a < b, a <= b, a > b; // boost by <=>.
    a == b, a != b; // boost by ==
    return 0;
}
```

Operator Overloading

- `<=>` actually returns `std::strong_ordering`, `std::weak_ordering` or `std::partial_ordering`, defined in `<compare>`.
 - For `std::strong_ordering`, it means only one of `a > b`, `a < b`, `a == b` will be `true`, and the other two is `false`.
 - E.g. integer comparison.
 - For `std::partial_ordering`, it means three of them may all be `false`.
 - E.g. float comparison, `NaN >/</==` is all `false`.
 - The ordering can be compared with 0.
 - Relation with 0 is the underlying relation, e.g. `<0` means `<`.
 - You can also use `bool std::is_lt/eq/gt/lteq/gteq/eq(ordering)` to judge what it is.
 - You cannot use it in `switch` since it's not integer.
 - `std::weak_ordering` is not that useful.
 - You may think two things are not equal strictly but make `== true` is this ordering, e.g. case-insensitive string.

Operator Overloading

- For `std::partial_ordering`, there is also an `unordered` relation; when all relations get `false`, then it is unordered.
 - Equal is in fact called **equivalent** here since `-0 == 0` while they are different in binary.
- You may notice that `<=>` is enough to know `==`, so why do we need to overload it manually?
 - For container, `<=>` is comparing every element one by one...
 - E.g. ms STL: (we'll cover lexicographical compare in *Algorithms*.)
- However, `==` may be cheaper since it can check size first.

```
std::vector v1{ 1,2,3 }, v2{ 1,2,3,4 };
auto r = v1 <=> v2;

1918
1919     return _Left.size() <=> _Right.size();
1920 } else {
1921     return _STD lexicographical_compare_three_way(_Left._Unchecked_begin(), _Left._Unchecked_end(),
1922     _Right._Unchecked_begin(), _Right._Unchecked_end(), _Synth_three_way{});
1923 }
```

Operator Overloading

- Besides, even if `a < b` where in fact only `b.operator<=>(B{a})` is valid, the compiler will not report errors since it will try `b > a` automatically!
 - This is not true for e.g. `operator+`, so you have to define it globally.
- Finally, if you just want to compare members one by one, then you can make comparison operators default like:

```
auto operator<=>(const Person&) const = default;  
bool operator==(const Person&) const = default;
```

- The `<=>` result is the first comparison result that makes `== false`; and it all `==` are `true`, then it's just equal.
 - The return type is the weakest ordering (strong->**weak**->partial), e.g. if there is a `double` as data member, it will return `std::partial_ordering`.

Operator Overloading

- `operator()`: you may accept any form of parameters; the object then act as if a function.
 - This object is called **function object** or **functor**.
- `operator[]`: you may accept one parameter; from the view of programmer, it's just like an array, which is given an index and give out the content.
 - It's really annoying that it can only accept a single parameter when you write a multi-dimension array.
 - You may write things like array of array as 2D-array so that `[]` can be used twice, but this will make the space not continuous, which hurts cache locality and performance.

Operator[]

- Since C++23, this is finally solved! You can use multidimensional subscript in `operator[]` like:
- This is achieved by re-explaining the comma expression in `[]`; that is, commas are regarded as separators of parameters.

```
class MDarr
{
public:
    MDarr(int initRow, int initCol) : row{initRow}, col{initCol}{ }
    float& operator[](int i, int j){ return arr[i * row + j];}
    const float& operator[](int i, int j) const { return arr[i * row + j];}
private:
    int row, col;
    float arr[100]; // not dynamic here.
};

int main()
{
    int rows = 2, cols = 3;
    MDarr arr{rows, cols};
    for(int row = 0; row < rows; row++)
        for(int col = 0; col < cols; col++)
            arr[row, col] = row * row + col;
    std::cout << arr[1, 2];
    return 0;
}
```

Operator Overloading

Though it's `static`, it's not necessary to call like `S::operator()(...)`; instead, `S{ }(...)` will be optimized if ctor is default so that no actual object will be constructed.

- Besides, `[]` and `()` can be static methods since C++23:
 - Many functors don't involve themselves in `operator()`, e.g. `less` than.
 - However, since it's in a class, you have to pass a `this`.
 - Unnecessary performance loss!
 - Now, by making it `static`, there won't be `this`.
 - `operator[]` is for uniformity since it can accept multiple parameters just like `operator()`.
 - These two operators are not that different now...
 - Final word: we'll improve these operators in *Move Semantics*.
- However, you may still be disturbed by defining lots of `CMP` class and `operator()` for e.g. `std::sort` before, though this `CMP` will **only be used here** and once...

Lambda Expression

- Since C++11, lambda expression is added as “temporary functor” or **closure**.
 - Beyond normal functions, lambda can capture something in the current context.
 - It is just an anonymous **struct** generated by the compiler, with an **operator() const**.

```
main::{lambda()#1}::operator()() const:
```
- Basic format: **auto func = [captures](params)->ReturnType {function body};**
 - Captures are actually members of the **struct**; you can also declare new variables.
 - ReturnType, params and function body are for **operator()**.
 - **->ReturnType** can be omitted when it can be deduced from **return**, just like **auto**.
 - Every lambda expression has its unique type.
 - **()** can be omitted if no parameter is passed.

Lambda Expression

- For example:
- Here we capture **carry** by **reference**, so modifying it in the function body will also change **carry** outside.
 - If you want to capture “all context” by reference, you may just use a **&**.
 - In fact the compiler only captures used variables.
- We can also capture variables by **copy**.
 - Deleting ampersand.
 - Read-only, unmodifiable.
 - If you want to capture “all context” by copy, you may use a **=**.
- You may also combine them, e.g. **[=, &carry]** or **[&, carry]**
 - Meaning that capture **carry** by ref/copy while others as copy/ref.

```
int carry = 0;
auto AddCarry = [&carry](char num) {
    if (num > '9')
    {
        num -= 10;
        carry = 1;
    }
    else
        carry = 0;
    return num;
};

// Declare new var:
auto AddCarry2 = [=, c2 = carry + 1](char num) {
```

Lambda Expression

- Note0: You should usually make a lambda expression short, otherwise it's possibly better to write a function.
 - For readability and avoidance of accidentally modifying some variables/ copy too many things/ omitting lifetime problem, it's recommended by Scott Meyer to not use `=/ &`, but write the used variables explicitly.
- Note1: capture by ref is just making the member `T&`, and by copy is makes it `T`.
 - Params captured by copy are also direct initialized.

Lambda Expression

- Note2: static and global variables don't need capture.
- Note3: If you use lambda expression in a non-static class method and you want to access all of its data members, you may need to explicitly capture **this** (by reference, since only copy pointer) or ***this** (really copy all members).
 - Capturing all by **=** and **&** can omit it.
 - Since C++20, it's **forced** to capture **this** explicitly even if there is **=**.
 - Also, lambda can access all members, including **private**.
 - If you just want to capture some of data members, you can capture them directly rather than **this**.

```
[&normals, this](size_t id)
{
    auto v1 = triangles[id][0], v2 = triangles[id][1],
        v3 = triangles[id][2];
    glm::vec3 e1 = vertices[v1] - vertices[v2],
        e2 = vertices[v2] - vertices[v3];
    normals[id] = glm::normalize(glm::cross(e1, e2));
});
```

triangles and **vertices** are data members.

Lambda Expression

- Note4: You may add specifiers after `()`.
 - `mutable`: since C++17, remove `const` in `operator()`, i.e. for capture by value, you can modify them (but don't really affect captured variable).
 - `static`: since C++23, same as `static operator()` (it's always better to use it if you have no capture!).
 - `constexpr/constexpr/constexpr`: we'll introduce them in the future.
- Note5: It's also legal to add attributes between `[]` and `()`.
 - Since C++23, if `()` is omitted, it's legal to add attributes and specifiers.
- Note6: function can also be written as `auto Foo(params) -> ReturnType { function body; }`, just like lambda expression!
 - Lambda expression is basically another form of function after all evolution since C++11...

You may wonder how a lambda can be non-static since you cannot get `this` of the anonymous `struct`. We'll cover it in *Move Semantics*.

Basics

Improvement In Execution Flow

Basic Control Flow

```
if (condition1) {  
    // ...  
}  
else if (condition2) {  
    // ...  
}  
else {  
    // ...  
}
```

```
while (condition1)  
{  
    // ...  
}
```

```
do {  
    // ...  
} while (condition1);
```

```
for (int i = 0; condition1; i++) {  
    // ...  
}
```

```
while (char ch = 1)  
    std::cin >> ch;
```

```
int integer = 0;  
switch (integer)  
{  
    case 0:  
        // ...  
        break;  
    case 1:  
        // ...  
        break;  
    default:  
        break;  
}
```

Basic Control Flow

- Particularly, you can switch enumeration since it's also kind of integer.

```
enum class BasicBufferType { OnlyColorBuffer, OnlyReadableDepthBuffer,  
                             ColorBufferAndWriteOnlyDepthBuffer, ColorBufferAndReadableDepthBuffer};
```

- **BasicBufferType::xxx** is the complete name, and it would be miserable to write it all in the switch statement!

- You may use **using enum** since C++20.

- Very fit for this use case!
 - In fact the whole name is **OpenGLFramework::Core::Framebuffer::BasicBufferType...**
 - Restrict it in the local scope.

```
switch (type)  
{  
    using enum BasicBufferType;  
    case OnlyColorBuffer:  
        GenerateAndAttachTextureBuffer_();  
        break;  
    case OnlyReadableDepthBuffer:
```

Basic Control Flow

- `case xx: {}`
 - `{}` is only necessary if you need to declare new variables in the scope.
- `break;`
 - Don't forget to break every `case`!
 - Sometimes you don't break deliberately, e.g. two cases do the same thing.
 - This is called **fall through**.
 - To show that you're deliberate and don't forget it (also disable compiler warning), you may use the attribute `[[fallthrough]]` since C++17.

```
case 0:
    [[fallthrough]];
case 1:
{
    // ...
    break;
}
```

Basic Control Flow

- Since C++20, `[[likely]]` and `[[unlikely]]` are added.
- You know many branch prediction strategies in ICS.
 - In *Computer Architecture*, you will learn more complex ones or even implement them yourself in Gem5.
 - One of the techniques is compiler-aided prediction; with labels in code, compilers may give hardware hints to take branch.
 - E.g. Add likelihood of loop unrolling, etc.
 - These two attributes are added for such optimization.
 - Usually, you only need to add them in hot spots found by profiling; otherwise it's very likely to reduce performance (compilers / processors are smart enough!).

```
if (flag) [[likely]] {  
    ...  
}
```

```
switch (num)  
{  
    [[likely]]  
    case 0:  
    {  
        // likely case..  
        break;  
    }  
}
```

```
for (int i = 0; flag; i++) [[likely]]  
    ;
```

Basic Control Flow

- You may want to give more hints than only branch prediction...
 - E.g. in loop unrolling, how many loops are combined together?
 - You've tried it in archlab in ICS...
 - If the compiler can *assume* loop times % 32 is always 0, it may choose 32 so that no trailing process is needed!
- Since C++23, you may use `[[assume(...)]]` to denote that!
 - You must ensure expression in assume is always **true**, otherwise UB.
 - So, use assume carefully! You shouldn't use them to check or document preconditions, but use them to **utilize known preconditions**!
 - There may also be many other possible optimizations, e.g. for signed integer, `[[assume(x >= 0)]]` may accelerate `x / 2` since positive ones can be `x >> 1` directly while negative ones need more concerns.
 - Particularly, `[[assume(false)]]` means here won't be reached, same as `std::unreachable()` in C++23 defined in `<utility>`.



```
[[assume(times % 32 == 0)]]  
for (int i = 0; i < times; i++)  
{  
    // ...  
}
```


Range-based for

- Since C++11, if a class specifies methods `begin()` and `end()`, you may use range-based for to iterate over it.

```
std::vector<int> v{ 1,2,4 };  
for (auto elem : v)  
    std::cout << elem << ' ';
```

- Similarly, this is equal to `elem = v[i];` as a copy; you need to use `auto&` if you want to ref it.
 - You can also use the exact type, e.g. `int` here.
- This structure is too limited...
 - What if I hope to drop the first element? filter by some condition?
 - This is extended since C++20 and we'll cover them in the future!
 - Also, we may uncover the essence of this loop in *Move Semantics*.

Code Block With_INITIALIZER

- You may write code like:

```
auto it = table.find(1);  
if (it == table.end())  
    ; // something  
else  
    ; // something
```

- But what if there are many **it** with different types in the code block?

- You have to specify **it1**, **it2**, ...
- But in fact, this **it** is just used once! It's a pollution to the name.

- Since C++17, you may code like:

- **it** is only valid in the if clause (including **else-if**, **else**)
 - You can also use **auto x = ..., y = ...;**
x != y if they're of the same type.

```
if (auto it = table.find(1); it == table.end())  
    ; // something  
else  
    ; // something
```

- This is kind of migration from **for**.
- Notice that if initialized variable can be converted to **bool** implicitly, then it's Okay not to write condition.

Code Block With_INITIALIZER

- This is also available in switch-clause since C++17.
 - You can also use e.g. `auto p = xx; p[0]` to switch `p[0]`.
- Since C++20, range-based for loop can also add an additional initializer, e.g. `for(auto vec = GetVec(); auto& m : vec);`
- Since C++23, type alias declaration can also be initializer, e.g. `if(using T = xx; ...)`.

```
switch (auto p = 1)
{
case 1:
```

Basics

Template

Template

BTW, Default template parameter is available, e.g.
`template<typename T = int>`, then `Func(...)` or `A<>{}`.

- We've also learnt some basic knowledge of template.
- Template is used to reduce redundant code, e.g. `Min(a, b)` for `float` or `int` shouldn't be duplicated.
- You can define a template function/class like:
 - `typename` can be substituted with `class` here.
 - You need to **instantiate** it with type parameter, e.g. `A<int> a; Func<float>();`.
 - Notice that `A<int>` and `A<float>` are different types!
 - For functions, types can be deduced from parameters.
 - e.g. `Func(1.0f)`; notice that return type cannot be deduced from **caller**, so if you have template parameter as return type, you "always need to specify explicitly.
 - Since C++17, **CTAD** (**class** template argument deduction) is introduced, meaning that argument of ctor can deduce the template parameter of class.
 - E.g. `std::vector v{1,2,3,4}` is enough.
- You can also define a template function as member.

```
template<typename T>
class A
{
    void Func(const T&) {}
};

template<typename T>
void Func(const T&) {};
```

```
template<typename T>
class A
{
    template<typename U>
    void Func(const U&) {}
};

class B
{
    template<typename T>
    void Func(const T&) {}
};
```

Template

Since C++14:

```
auto less = [] (const auto& a, const auto& b) static { return a < b; };
```

C++23

Since C++20:

```
auto less = [] <typename T> (const T& a, const T& b) static { return a < b; };
```

- Lambda expression can also use template:
- This is same as an anonymous struct, with **static operator()** that is a template function.
 - You can e.g. pass the functor to algorithms in **<algorithm>** (which will be covered in the following two lectures).
- Since C++20, abbreviated function template is introduced.
 - Every **auto** implicitly means a new template parameter, so you can also e.g. **Func<int>**.
 - This is different from generic lambda expression, since the functor itself is not template, but its **operator()** is. You can pass the functor as parameter, but template cannot (and needs instantiation!).
 - It's **strongly recommended** not mixing abbreviated template with **template<typename T>**, which will cause many subtle problems.

```
void Func(const auto& v) {}
```

If you're really interested in why, you can check *C++20 – The complete guide* 2.3.2 after learning advanced template knowledge in our future lectures.

Summary

- auto, const auto, auto&, ...
- Integers, floating points, pointers, references, array, function, enumeration
- Type alias, attributes
- Expressions, evaluation order.
- Class, member functions (including special ones like ctor & dtor), object initialization, access control, inheritance; struct, bit field.
- Function overloading, functor, lambda expression.
- if(auto), range-based for, notes on switch-case.
- Template
- There are many important but basic concepts, so we deliberately review what you've learnt with new knowledge to make it easier.
 - You may also review all content yourself to digest after this lecture!

Next lecture...

- We'll cover "STL", i.e. iterators, containers.
 - Allocators, if you've heard of them, will temporarily not covered here.
- Since you've learnt basic knowledge in 程设&数算, we'll teach their implementation for you to understand some regulations deeply.
- The content will be massive, ready for it!