Learning Notes

Huang Bo

Exported on May 17, 2023

Table of Contents

1 虚函数 3

2 Merge Tool Usage/合并操作 6

3 Vector和Linked-List 11

4 Useful Shortcut of Visual Studio 13

5 C++ Performance and Optimization 16

6 Header File Dependency Optimization 19

7 Basic Debug Knowledge 21

7.1 Breakpoint 21

7.2 Dump File 23

8 Some Best Practice of Code Design 25

8.1 Use assert 25

8.2 Avoid hard code 26

# 虚函数

**虚函数**

虚函数是c++提供的用于实现OPP多态特性的机制。

当使用一个基类的指针或者引用调用一个虚函数时，实际调用的函数需要运行时才能确定，因此不同的派生类可以执行不同的操作。

**使用规则**

在基类中使用virtual声明虚函数，在派生类中可以使用override显式声明其覆盖了基类的虚函数。

注意两者的形参必须匹配，是否为const也必须匹配。如果不匹配且没有使用override，那么会声明一个新的非虚函数。

Example:

|  |
| --- |
| **class** Base  {  **public**:  **virtual** int func();  };  **class** D1: **public** Base  {  **public**:  int func(int); //声明了一个新的函数  };  **class** D2: **public** D1  {  **public**:  int func(int); //非虚函数，隐藏了D1：：func(int)  int func(); //override了Base的虚函数  };  **class** D3 : **public** D2  {  };  D1 d1; D2 d2; D3 d3;  Base \*p1 = &d1, \*p2 = &d2, \*p3 = &d3;  p1->func(); //Base::func()  p2->func(); //D2::func()  p3->func(); //D2::func() |

还可以使用final来阻止之后的派生类覆盖该虚函数，注意此时仍然会继承该虚函数。

虚函数也可以包含默认实参，但是调用时以当前的静态类型为准。故基类和派生类的默认实参应保持一致。

当某个函数对于当前基类来说没有实际意义，只作为派生类接口的声明时，可以将其声明为纯虚函数。

在虚函声明的最后加上=0则可以将其声明为纯虚函数。

含有纯虚函数的类称为抽象基类。抽象基类无法定义具体对象。

Example:

|  |
| --- |
| **class** Shape  {  **public**:  **virtual** float calculateArea() = 0;  **virtual** ~Shape() {}  };    **class** Rectangle : **public** Shape  {  **public**:  Rectangle(float w, float l) :w(w), l(l) {}  **virtual** float calculateArea() override { **return** w \* l; }  **private**:    float w, l;  };    **class** Circle : **public** Shape  {  **public**:  Circle(float r) :r(r) {}  **virtual** float calculateArea() override { **return** pi \* r\*r; }  **private**:  float r;  }; |

如果调用时不想进行动态绑定，可以使用作用域运算符： baseObject->Base::func(); 通常用于在派生类的虚函数里调用其基类版本时。如果不加作用域运算符，会导致自身被递归调用。

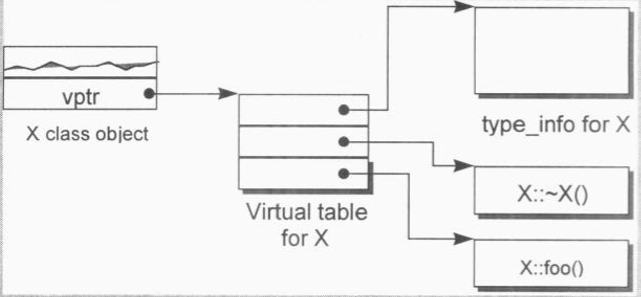
|  |
| --- |
| **class** Square : **public** Rectangle  {  **public**:  Square(float a) : Rectangle(a, a) {}  **virtual** float calculateArea() override { **return** **this**->Rectangle::calculateArea(); }  }; |

基类通常需要定义一个虚析构函数，这样才能在动态分配对象时，确保delete会调用到正确的析构函数。

|  |
| --- |
| **class** Shape  {  **virtual** ~Shape() {}  }; |

**内存模型**

含有虚函数的类，在内存里会生成一个vtable，存放当前类所有虚函数的地址。而每一个实例对象会生成一个vptr，存放vtable的地址。

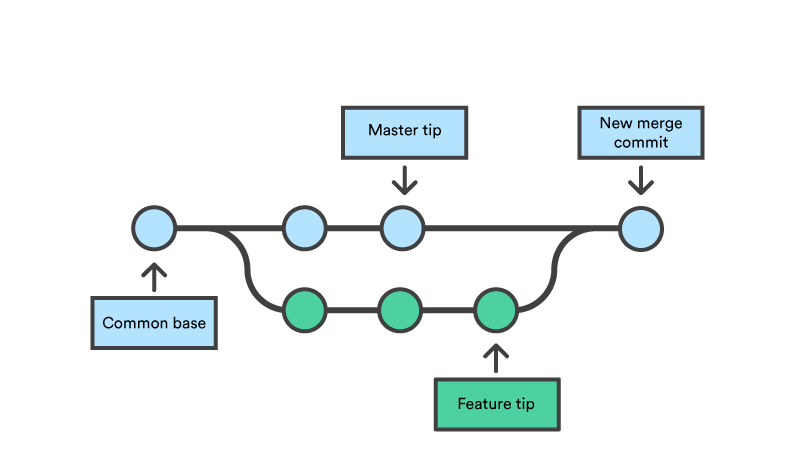


派生类继承基类时会复制vtable里的内容。当派生类有override基类的虚函数，则会将vtable里相应的地址修改为派生类虚函数的地址，这样当派生类的对象调用该函数时就能找到正确的虚函数。

# Merge Tool Usage/合并操作

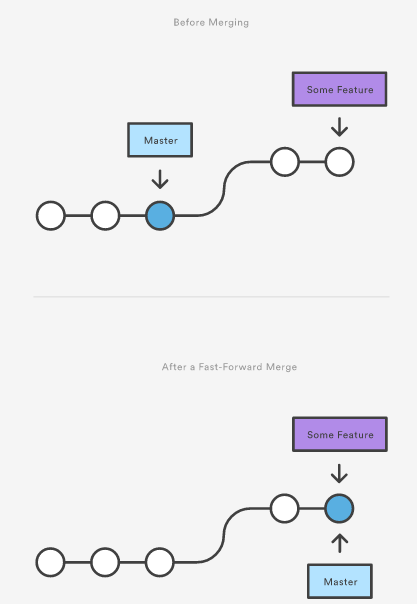
merge指的是git里将两个分支合并的操作，通常是将feature branch合并回master branch。

merge操作是将其他分支合并到当前分支，使用前注意要checkout到正确的分支。



**Fast Forward Merge**

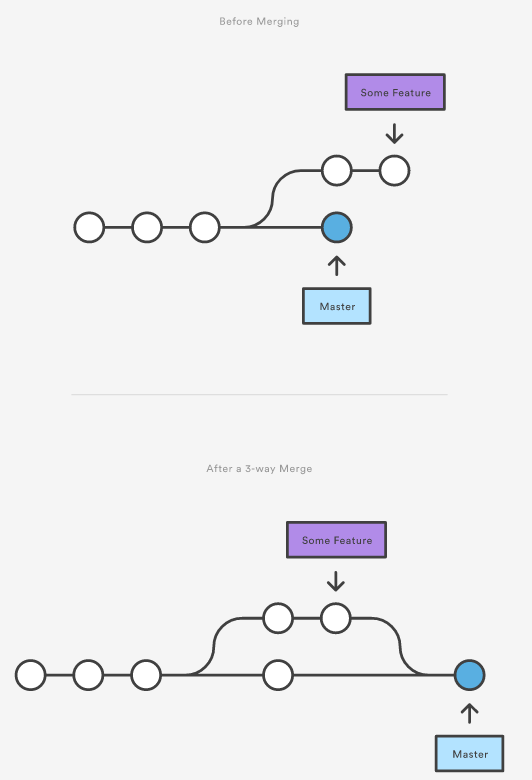
如果merge时候当前分支可以向前直达目标分支，两者没有冲突，此时git会直接将分支标记移动到目标分支来快速merge，避免commit。



如果想保留分支信息，可以关掉该选项来创建一个新的commit。

**Three Way Merge**

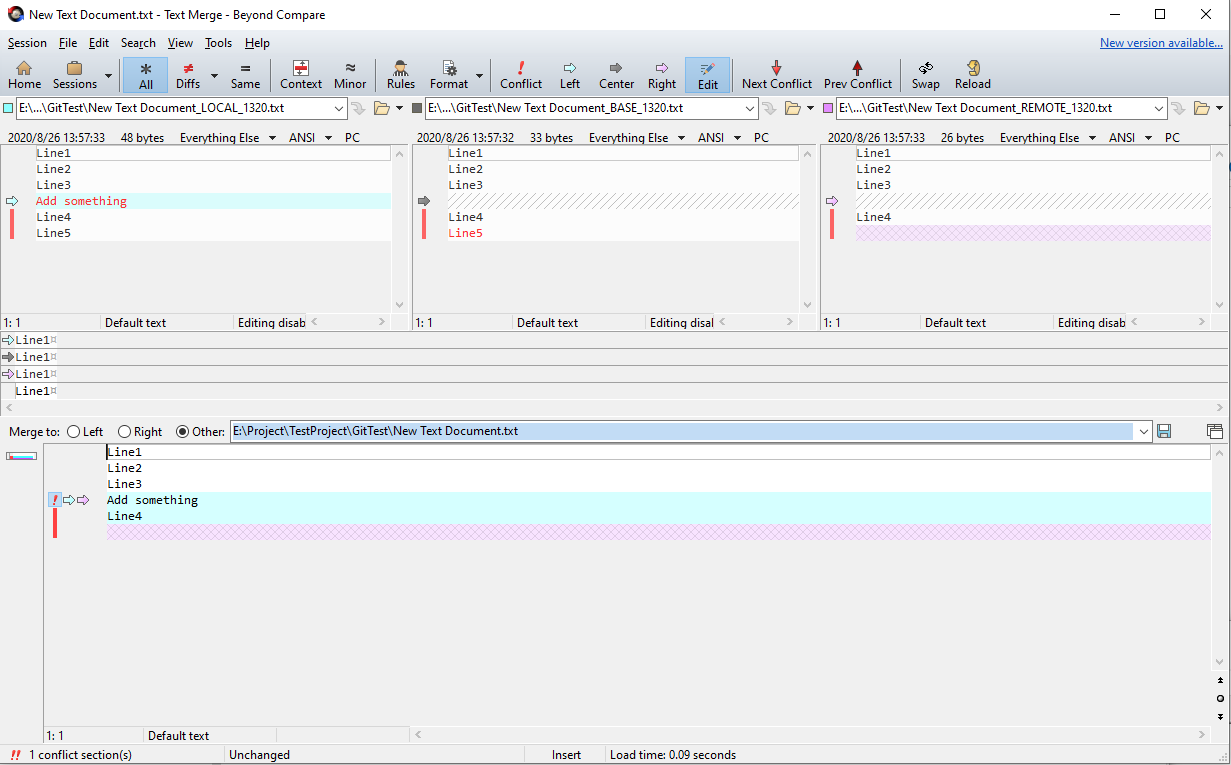
如果merge时当前分支无法向前直达目标，则需要进行Three Way Merge。



如果两个分支都修改了同一个文件的相同区域，则git自动merge会失败，会报出conflict，需要手动合并。

使用merge tool可以方便的查看合并时的信息。

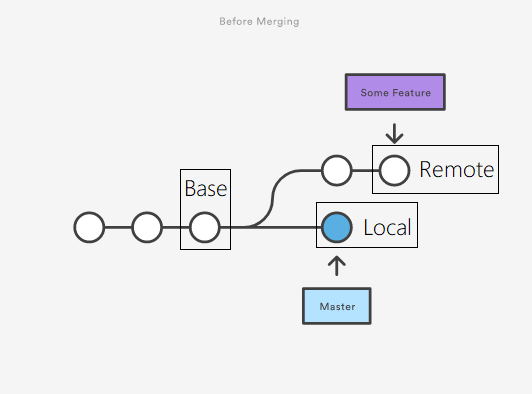
以BeyondCompare为例：



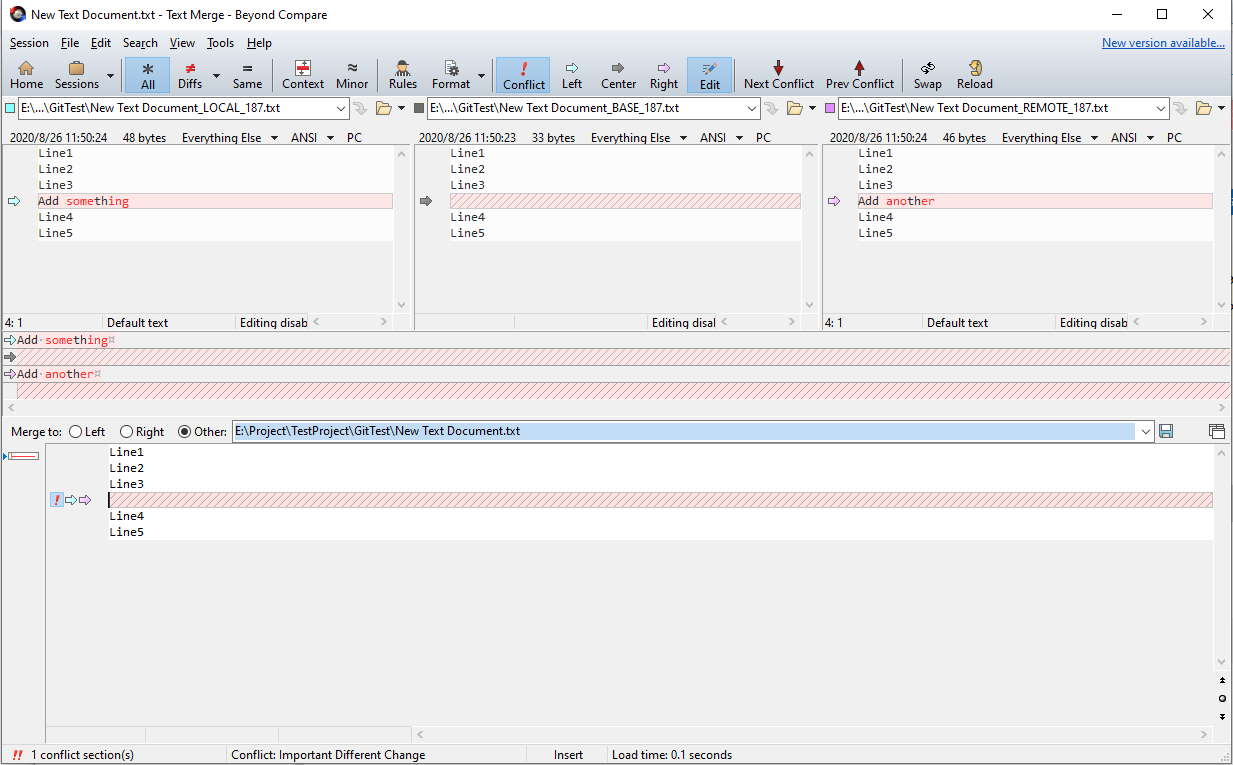
中间的Base表示当前文件两个分支共同的前置版本。

左侧Local表示当前分支的改动。

右侧Remote表示目标分支的改动。



手动合并时要特别注意两个改动有重复部分的情况，此时merge工具不能自动得出正确的结果，需要手动操作。



# Vector和Linked-List

**std::vector**

大小可变的数组。使用连续的内存存放数据，随机访问速度很快。在尾部以外的位置插入/删除很慢。

初始化  
初始化时会动态分配一块内存，用于存放数据。  
可以调用reserve来指定分配的大小，来避免之后添加元素时重新分配内存。

插入/删除元素  
尽量在末尾插入/删除，在头部和中间操作数据时，需要将之后的数据全部移动，速度会很慢  
当预先分配的内存占满时执行添加操作，会进行扩容操作，一般会扩容2倍。扩容将分配新的内存，并将原数据复制过去，之后释放原内存。

访问元素  
可以通过下标[]来访问已存在的元素。注意，不能通过该方式添加元素。  
可以使用迭代器来循环访问，但是在添加和删除元素后，原有的迭代器可能会指向错误的元素，故尽量不要在循环内改变容器的大小。

**std::list**

双向链表。使用指针记录前后数据的位置，只能双向顺序访问数据。在任何位置插入/删除的操作速度都很快。

初始化  
list以node作为存放单个数据的结构，还包含指向前后元素的指针。

插入/删除元素  
插入/删除元素时，会构造/析构一个节点，并调整前后节点的指针。  
插入操作不会导致原有的迭代器失效。删除操作会导致被删除元素的迭代器失效。  
list内部也支持移动某个范围的元素，以该范围为基础修改节点指针的值。  
可以在两个相同类型的list之间移动元素。  
使用emplace来避免构造临时的对象。  
  
自己实现的vector，list及相关算法  
<https://github.com/douiwby/DataStructureExercise/tree/master/DataStructure/Header>

**UE4的TArray**

UE4提供三种容器TArray，TMap和TSet，分别对应了std::vector,std::map(std::unordered\_map),std::set(std::unordered\_set)  
存放在容器内的UObject会添加一个引用，避免被Garbage Collection回收。  
可以提供自定义的Allocator来控制内存分配，如果没有提供，会使用默认的FHeapAllocator

TArray提供了一系列与std::vector类似接口，来操作容器。  
添加元素的接口：Add,Emplace,Append,AddUnique,Insert等  
删除元素的接口：Remove,RemoveAt,RemoveSwap,RemoveAtSwap,Empty等  
改变大小的接口:SetNum,Shrink等  
查询容量的接口：Num,Max,GetSlack(获取剩余容量，等于Max-Num)等

可以使用GetData()来直接对底层数据进行操作

例子：  
TArray<FString> UninitStrs;  
UninitStrs.Emplace(TEXT("A"));  
UninitStrs.Emplace(TEXT("D"));  
UninitStrs.InsertUninitialized(1, 2);  
new ((void\*)(UninitStrs.GetData() + 1)) FString(TEXT("B"));  
new ((void\*)(UninitStrs.GetData() + 2)) FString(TEXT("C"));  
// UninitStrs == ["A","B","C","D"]

# Useful Shortcut of Visual Studio

You can find all shortcuts of latest Visual Studio here:  
<https://docs.microsoft.com/en+us/visualstudio/ide/default+keyboard+shortcuts+in+visual+studio>

Here are some useful and most common shortcuts.

**General Text Editor**

|  |  |
| --- | --- |
| Ctrl+X/Ctrl+C/Ctrl+V | If without selection it will handle the entire line. |
| Ctrl+L | Cuts all selected lines or the current line if nothing has been selected to the clipboard. |
| Ctrl+Z/Ctrl+Y | Undo/Redo. |
| Ctrl+S/Ctrl+Shift+S | Saves one/all files. |
| Home/End/Ctrl+Home/Ctrl+End | Move cursor to the start/end of this line/file. |
| Ctrl+Right Arrow/Ctrl+Left Arrow | Moves the cursor one word to the right/left. |
| Ctrl+PageUp/Ctrl+Pagedown | Moves cursor to the top/bottom of this page. |
| Shift+(move cursor shortcuts) | Select the cursor moved section. |
| Ctrl+A | Select all. |
| Ctrl+Up Arrow/Ctrl+Down Arrow | Scrolls text window up/down one line but does not move the cursor. |
| Ctrl+Enter/Ctrl+Shift+Enter | Inserts a blank line above/below the cursor. |
| Ctrl+U/Ctrl+Shift+U | Changes the selected text to lowercase/uppercase characters. |
| Ctrl+Backspace/Ctrl+Delete | Deletes the word to the left/right of the cursor. |
| Ctrl+Tab | Cycles through all tabs. |
| Ctrl+Shift+V | Cycle through the list of clipboard contents. |
| Ctrl+R, Ctrl+W | Shows or hides spaces and tab marks |

**Code Editor**

|  |  |
| --- | --- |
| Tab/Shift+Tab | Moves current line or selected lines one tab right/left |
| Ctrl+K, Ctrl+C / Ctrl+K, Ctrl+U | Marks the current line or selected lines of code as a comment/uncomment. |
| Ctrl+Spacebar/Alt+Right Arrow | Displays statement auto completion(Edit.CompleteWord). |
| Ctrl+J | Lists members for statement completion(Edit.ListMembers). |
| Ctrl+R, Ctrl+R | Refactor.Rename |
| Ctrl+K, Ctrl+X | Insert code snippets in current position. |
| Tab | Enter snippets name and press Tab will get full code snippets. |
| Ctrl+K, Ctrl+S | Surround select code with code snippet. |
| Ctrl+K, Ctrl+D | Auto format whole document.(Edit.FormatDocument) |
| Ctrl+K, Ctrl+F | Auto format selection part.(Edit.FormatSelection) |

**Code Navagation**

|  |  |
| --- | --- |
| Ctrl+G | Go to specific line. |
| F12/Ctrl+F12 | Go to the definition/declaration for the selected symbol in code |
| Ctrl+-/Ctrl+Shift+- | Moves backward/forward in the navigation history. |
| Ctrl+] | Find the matching brace("**{}**"). |
| Ctrl+M, Ctrl+M | Toggles the currently outlining between hidden and display states. |
| Ctrl+M, Ctrl+L | Toggles all outlining between hidden and display states. |
| Ctrl+M, Ctrl+O | Collapse all functions.(Edit.CollapseToDefinitions) |
| Ctrl+F/Ctrl+H | Displays the Find/Replace dialog. |
| Ctrl+Shift+F/Ctrl+Shift+H | Displays the Find/Replace in Files dialog. |
| F3/Shift+F3 | Finds the next/previous occurrence of the previous search text |
| Ctrl+F3/Ctrl+Shift+F3 | Finds the next/previous occurrence of the currently or selected word. |

**Debug**

|  |  |
| --- | --- |
| F9 | Sets or removes a breakpoint at the current line |
| Ctrl+F9 | Enables or disables the breakpoint on the current line of code. |
| Ctrl+B | Opens the New Function Breakpoint dialog. |
| Ctrl+Alt+Q | Displays the Quick Watch dialog. |
| F5/Ctrl+F5 | Start with/without debugging. |
| Ctrl+Shift+F5 | Restart debugging. |
| Shift+F5 | Stop debugging. |
| Ctrl+F10 | Starts or resumes execution of your code and then halts execution when it reaches the selected statement.(Debug.RunToCursor) |
| Ctrl+Shift+F10 | Sets the execution point to the line of code you choose.(Debug.SetNextStatement) |
| Alt+NUM \* | Show the next statement to be executed. |
| F10 | Executes the next line of code but does not step into any function calls.(Debug.StepOver) |
| F11 | Executes code one statement at a time,  tracing execution into function calls.(Debug.StepInto) |
| Shift+F11 | Executes the remaining lines of a function in which the current execution point lies.(Debug.StepOut) |

**Build**

|  |  |
| --- | --- |
| Ctrl+Shift+B | Build solution. |
| Ctrl+Break | Cancel the build. |
| Ctrl+F7 | Compile current file. |

# C++ Performance and Optimization

Before do practice, you should know which code you should take most care of.

There is a quote: make the common case fast and the rare case correct. That means the frequently used code should optimized most. If you know your function will be called frequently, you may need consider it's performance and do optimization early.

Pareto's law(二八法则) also works for program. Use profile to find where time spend, optimize the code that take the most time. It's not worth to optimize code that take little time.

Here are some best practices that help to improve performance.

**Statement**

Moving declarations outside loops to avoid extra construction and destruction.

Avoid function calls inside loop. For example, "*for(...) DoSomething();*" should change to "*DoSomething() { for(...){...} }*".

Consider use *switch* to replace long *if...else if...* statement and *operator?:* . If not possible, put the most common case at the beginning.

When use switch, the constant value used in case better have no gaps or few gaps. In this way, compiler can generate a jump table to make code more efficient. If the gaps is too big, it may just like normal *if...else if...* statement.

Also, put the most common case at the beginning when use switch.

For most classes, use the operators+= instead of operators+ to avoid create temporary object.

For objects, use the prefix operator(++obj)instead of the postfix operator(obj++).

Pass structures by reference, pass fundamental types by value.

When you pass an object as argument, if it may be null, pass by pointer, otherwise, pass by reference.

**Function**

Use inline for short functions.

Reduce the number of function parameters and local variables if possible.

If function has more arguments than available registers, they will be placed into stack. You can group arguments to one class, so register have a chance to be used. This also works for local variables.

To avoid costly conversions, define overloaded functions for the most typical argument types. Such implicitly conversions will create a temporary object of the expected type.

**Class**

Make default constructors as lightweight as possible, cause there may be many temporary object created.

Use initialization instead of assignment to avoid call default constructor. Example: Use *Color c(black);* rather than *Color c; c = black;* .

Declare as explicit all constructors that receive only one argument, except for the copy constructors of concrete classes.

Declare member function as static if possible. In they way, the implicit this argument is not passed.

Only declare virtual when the class is intended for use as a base class, and only declare virtual function when you intend to override them. Due to an additional virtual table, classes that have virtual member functions occupy more storage space, their construction requires more time, and every virtual member function has a slower call time.

**Container**

When deal with two and higher dimensional arrays, loops the most inner first. Because a[i][j] and a[i][j+1] is adjacent, but not for a[i+1][j]. There is a good chance that CPU can read adjacent data which already in cache rather than load a new block of memory.

When you want to use equivalent expressions of STL container, choose the more general expressions, they are available for more containers. Example: call *a.empty()* instead of *a.size() == 0*, call *iter != a.end()* instead of *iter < a.end()*, and call *distance(iter1, iter2)* instead of *iter2 - iter1*.

Use of container member functions instead of common algorithm. Example: *std::set::find*.

Use binary search algorithms on sorted sequence.

Before adding elements to a vector or to a string object, call its member function reserve with a size big enough for most cases.

**Thread**

In an interactive application, whenever you must perform an operation that can take more than few seconds, assign the operation to a worker thread having a lower priority than normal.

In a multi-core system, if you can split a CPU-intensive operation across several threads, use as many worker threads as there are processor cores.

**I/O**

Store text files in a compressed format. Disk have much less bandwidth than processors. By (de)compressing on the fly, the CPU can speed up I/O.

Instead of storing data in text mode, store them in a binary format. Binary numbers occupy less space, and it is faster to transfer them between memory and disk.

Instead of opening and closing an often needed file every time you access it, open it only the first time you access it, and close it when you are finished using it.

**Math**

Consider ways of rephrasing your math to eliminate expensive operations.

Avoid sqrt() especially in comparison.

Pre-compute expressions, and reuse the results if you need use that for multiple times.

If you repeatedly divide by x, consider computing 1/x and multiplying by the result.

Try to simplify your equations on paper first.

**Others**

1.Try to avoid casting where possible.

Integer and floating point instructions often operate on different registers, so a cast requires a copy.

Shorter integer types (char and short) still require the use of a full-sized register, and they need to be padded to 32/64-bits and then converted back to the smaller size before storing back in memory.

2.Access memory in increasing addresses order. In particular:

    scan arrays in increasing order;

    scan multi-dimensional arrays using the rightmost index for innermost loops;

    in class constructors and in assignment operators (operator=), access member variables in the order of declaration.

3.Keep the compiler default memory alignment.

4.Declare in an anonymous namespace the variables and functions that are global to a compilation unit, but that are not used by other compilation units.

5.If in a class template, there is function part that does not depend on any template parameter, define a non-member function having this part, and replace the original function body with a call to the new function. This will avoid duplicated machine code of this part when there are multiple instances.

6.If you often have to call a pure function that has a small integer interval as domain, pre-compute all the values of the function for every value of the domain and put them in a static array called lookup table. When you need the value of the function for a particular value of the domain, read the corresponding value of such array.

Example:

*double sqrt10(int i)*

*{*

*static const double lookup\_table[] = {*

*0, 1, sqrt(2.), sqrt(3.), 2,*

*sqrt(5.), sqrt(6.), sqrt(7.), sqrt(8.), 3,*

*};*

*assert(0 <= i && i < 10);*

*return lookup\_table[i];*

*}*

7.If you often have to call a pure function with arguments that in most cases belong to a small domain, use a static map (aka dictionary) that is initially empty. When the function is called, search the map for the function argument. If you find it, return the associated value, otherwise compute the result and insert the pair argument-result into the map.

8.Try to use hardware accelerators, like SIMD, CUDA, c++ AMP etc.

9.Try to use profile guided optimization

# Header File Dependency Optimization

How header file affect project?

When compile a project, the compiler will replace #include pragma with content of header files.

Here are two bad points that may happen if programmer didn't take care about header file dependency:

When change one content of one header file, all related obj need to be regenerated. If one cpp file include header which it didn't actually need, it may get compiled more often than it should be.

If one header include another header on by one, the compiler will proceed more content, this will increase the compile time, especially when lots of cpp include this header.

How to optimize according to these two points?

Point1: Only include files that you really need. Don't just include a huge header for convenient. For example, if you only need class A and class B, include A.h and B.h, do not include another header that contain A.h B.h, and C.h.

Point2: Use forward declaration.

Forward declaration is a way to declare the name without it's definition. It's helpful for two class which contain each other. But we are not talking that case, we can use forward declaration when we only need the pointer of some member class. In this case, we don't need include its header file, which reduce the files that compiler proceed, also there will be less files need recompile when you change the content of this class.

Here is a example:

If the code like this:

#include<SomeClass.h>

class AnotherClass

{

...

    SomeClass mSomeClass;

...

};

you can replace the mSomeClass with pointer and remove the include like:

//#include<SomeClass.h>  //We don't need this with forward declaration

class SomeClass;

class AnotherClass

{

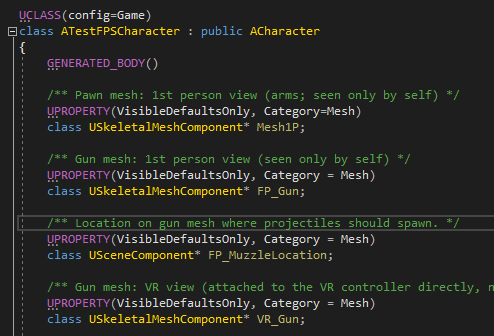
...

    SomeClass\* mSomeClass;

...

};

Here is the example in UE4 sample:



# Basic Debug Knowledge

## Breakpoint

Breakpoints can pause debugger execution. You can set breakpoints if you want to see the state of code variables or look at the call stack at a certain breakpoint.

**Set breakpoints in source code**

Simply press F9 to set a breakpoint at current line.

**Breakpoint conditions**

You can control when and where a breakpoint executes by setting conditions.

You can choose Conditional Expression, Hit Count, or Filter, and set the value accordingly.

**1.Conditional expression**

When you select Conditional Expression, you can choose between two conditions: Is true or When changed. Choose Is true to break when the expression is satisfied, or When changed to break when the value of the expression has changed.

Note: conditional expression is slow, don't put it in a loop. You can write if code in loop and set breakpoint inner to avoid it slow down program.

**2.Hit count condition**

If you suspect that a loop in your code starts misbehaving after a certain number of iterations, you can set a breakpoint to stop execution after that number of hits, rather than having to repeatedly press F5 to reach that iteration.

**3.Filter condition**

You can restrict a breakpoint to fire only on specified devices, or in specified processes and threads.

Under Conditions in the Breakpoint Settings window, select Filter, and then enter one or more of the following expressions:

* MachineName = "name"
* ProcessId = value
* ProcessName = "name"
* ThreadId = value
* ThreadName = "name"

Enclose string values in double quotes. You can combine clauses using & (AND), || (OR), ! (NOT), and parentheses.

**Function breakpoints**

You can break execution when a function is called. This is useful, for example, when you know the function name but not its location. It is also useful if you have functions with the same name and you want to break on them all (such as overloaded functions or functions in different projects).

Shortcut Ctrl+B will create a function breakpoint.

To narrow the function specification:

* Use the fully qualified function name.  
      Example: Namespace1.ClassX.MethodA()
* Add the parameter types of an overloaded function.  
      Example: MethodA(int, string)
* Use the '!' symbol to specify the module.  
      Example: App1.dll!MethodA
* Use the context operator in native C++.  
      {function, , [module]} [+<line offset from start of method>]  
      Example: {MethodA, , App1.dll}+2

Don't forget to choose the right language of the function.

**Set a function breakpoint using a memory address**

You can use the address of an object to set a function breakpoint on a method called by a specific instance of a class. For example, given an addressable object of type my\_class, you can set a function breakpoint on the my\_method method that instance calls.

Example:

1. Set a breakpoint somewhere after the instance of the class is instantiated.
2. Find the address of the instance (for example, 0xcccccccc).
3. Select Debug > New Breakpoint > Function Breakpoint, or press Alt+F9 > Ctrl+B.
4. Add the following to the Function Name box, and select C++ language.  
       ((my\_class \*) 0xcccccccc)→my\_method

**Data breakpoints**

Data breakpoints break execution when a value stored at a specified memory address changes. If the value is read but not changed, execution doesn't break.

You can specific a memory address, or an expression that evaluates to a memory address. For example, type &avar to break when the contents of the variable avar changes, but take care of it's scope.

Data breakpoints are automatically disabled at the end of each debugging session. Because visual studio assume that the address of a variable changes from one debugging session to the next.

**Set a breakpoint in the Call Stack window**

To break at the instruction or line that a calling function returns to, you can set a breakpoint in the Call Stack window.

The call stack breakpoint appears in the Breakpoints window as an address, with a memory location that corresponds to the next executable instruction in the function.

**Set a breakpoint in the Disassembly window**

After program paused during debugging, you can open the Disassembly window and set breakpoint here.

**Manage breakpoints in the Breakpoints window**

You can use labels to sort and filter the list of breakpoints in the Breakpoints window.

To save or share the state and location of your breakpoints, you can export or import them.

## Dump File

What is dump

 A dump file is a snapshot that shows the process that was executing and modules that were loaded for an app at a point in time. A dump with heap information also includes a snapshot of the app's memory at that point.

What is dump used for

Opening a dump file with a heap in Visual Studio is something like stopping at a breakpoint in a debug session. Although you can't continue execution, you can examine the stacks, threads, and variable values of the app at the time of the dump.

Dumps are mostly used to debug issues from machines that developers don't have access to. You can use a dump file from a customer's machine when you can't reproduce a crash or unresponsive program on your own machine. Testers also create dumps to save crash or unresponsive program data to use for more testing.

**Dump files with or without heaps**

Dump files may or may not have heap information.

* Dump files with heaps contain a snapshot of the app's memory, including the values of variables, at the time of the dump. Visual Studio also saves the binaries of loaded native modules in a dump file with a heap, which can make debugging much easier. Visual Studio can load symbols from a dump file with a heap, even if it can't find an app binary.
* Dump files without heaps are much smaller than dumps with heaps, but the debugger must load the app binaries to find symbol information. The loaded binaries must exactly match the ones running during dump creation. Dump files without heaps save the values of stack variables only.

**Find .exe, .pdb, and source files**

To use full debugging features on a dump file, Visual Studio needs:

* The .exe file the dump was created for, and other binaries (DLLs, etc.) that the dump process used.
* Symbol (.pdb) files for the .exe and other binaries.
* The .exe and .pdb files that exactly match the version and build of the files at dump creation.
* Source files for the relevant modules. You can use the disassembly of the modules if you can't find the source files.

If the dump has heap data, Visual Studio can cope with missing binaries for some modules, but it must have binaries for enough modules to generate valid call stacks.

**Debug a dump file**

After open a dump in Visual Studio, the Minidump File Summary window shows summary and module information for the dump file, and actions you can take.

Select Debug to start debugging. Now it’s as if you had managed to set a break point at the exact moment of the stack overflow. This also presents the opportunity to review Call Stack, review other threads, and even verify the state of any Local variables at that point in time.

# Some Best Practice of Code Design

## Use assert

An assertion statement specifies a condition that you expect to be true at a point in your program. If that condition is not true, the assertion fails, execution of your program is interrupted and an error message is displayed.

**assert**

Assertion statements compile only if \_DEBUG is defined. Otherwise, the compiler treats assertions as null statements. Therefore, assertion statements impose no overhead or performance cost in your final Release program, and allow you to avoid using #ifdef directives.

Do not rely the result of the expression in assert, it won't execute on release build, example: assert(num++>0);

Assertions are mainly used to **check logically impossible situations**. You can set an assertion on a condition that must be true according to the logic of your program. The assertion has no effect unless a logic error occurs.

In C/C++, you can use the preprocessor NODEBUG to disable assertions.

**static\_assert**

The normal assert is executed at run time. C++ supports yet another form of assert known as the static\_assert and it performs compile-time assertion checking. It is present since C++11.

static\_assert ( bool\_constexpr , message )         (since C++11)

static\_assert ( bool\_constexpr )                         (since C++17)

If bool\_constexpr evaluates to false, then a compiler error is issued.

Unlike #error, assertion using static\_assert takes place after the preprocessing translation stage. Therefore, it is possible to check for the size of a datatype with sizeof using static\_assert.

Example:

static\_assert(sizeof(void \*) == 4, "64-bit code generation is not supported.");

static\_assert(sizeof(VMPage) == PAGESIZE, "Struct VMPage must be the same size as a system virtual memory page.");

It also can be used to detect common usage errors in template at compile time.

Example:

template<typename T, int Size>

void handleArray(T (&a)[Size])

{

    static\_assert(Size > 3, "Size is too small");

    ...

}

## Avoid hard code

Hard code is the practice of embedding data directly into the source code of a program or other executable object.It can cause some problems during coding.

It may has bad readability. A "magic number" could make people confused. A variable with reasonable name won't have such issue.

It's hard to modify. If you want to change the configuration writen as plain value, it's difficult to find all "reference". Use constant variable instead of the value.

if(currentTime>50) //Bad

If you are sure the value won't change, it's OK to use value.

if(fileExtension=="txt") //OK

It need rebuild once you want to change it. Store value that may often be changed in a configuration file can avoid this issue.

The best practice is using hard-coded data on unchanging pieces of information, such as physical constants, version numbers and static text elements. Softcoded data, on the other hand, encode arbitrary information like user input, HTTP server responses, or configuration files, and are determined at runtime.