14. Performance

14.1. **const** correctness

Using **const** tells the compiler that objects/variables should not change:

```
void function1(const std::string & str);   // Pass by reference-to-const

void function2(const std::string * sptr);  // Pass by pointer-to-const

void function3(std::string str);           // Pass by value, str unchanged
```

For the above to-const parameter functions, the C++ compiler checks whether the passed can be changed, or is passed further as a const. Example:

```
void mutate(std::string & s) {};

void function1(const std::string & str) {
  mutate(str);            // compiler error: str is const
  std::string localCopy = str;
  mutate(localCopy);  // fine, localCopy is not const
}
```

14.1. **`const`** correctness

Declaring the const-ness of a parameter is just another form of type safety and should be done as soon as they are declared.

When you declare something const, *the compiler will treat it as a* ***different type*** than the non-const version.

const overloading of methods or operators allows const correctness:

constExample01.cpp

```
class Item { /*...*/ };

class MyItemList {
 public:
   const Item & operator[] (int index) const;  // [] operators often have a
   Item & operator[] (int index);               // const and non-const version
   // ...
};
```

14.1. **const** correctness

const correctness allows:

1. protection from accidentally changing variables / objects
2. protection from making accidental variable assignments, e.g.:

```
void myMethod(const int x) {
  if ( x = y )   // typo: really meant if (x == y) -> error
  //  …
}
```

3. the compiler to optimize for it

More examples can be found here.

14.1. **const** correctness

Reminder: **const** pointers can come in various forms, what matters is that everything on the left of the **const** keyword is constant. If **const** is on the full left, what is on its right is constant. **const** pointers need to be directly initialized:

```
int myInteger = 71;
const int constInt = 17;
int const * pointToConstInt = &constInt;    // pointer to const int
int * const constPointToInt = &myInteger;   // const pointer to int
int const * const cPointToCInt = &constInt;  // const pointer to const int
const int * pointToConstInt2 = &constInt;   // pointer to const int
```

14.1. **const** correctness

Reminder: Example 03 from 7. Pointers (difficulty level: 🌶️🌶️🌶️):

```
/** Print a mouse in the console, using a const pointer to avoid changes */
#include <iostream>      // terminal output
[[nodiscard]] auto * getBitmapAddress() {
    static char bitmap[] = "(^._.^)~";  // "bitmap" created in static memory
    return bitmap;  // return pointer to first element
}
int main() {
  // using a pointer to bitmap, and incrementing it, is possible:
  auto * mousePointer = getBitmapAddress();
  while ( *mousePointer != 0 )  std::cout << *(mousePointer++);
  std::cout << "\n";
  // Here mousePointer has changed, it's hard to get the original pointer.
  // Modify the above by protecting the pointer with const and redo the loop.
}
```

14.2. **constexpr** – Generalized constant expressions

Since C++11, the **constexpr** specifier declares that the expression that follows is always evaluated at compile-time, and thus:
- can save potentially significant processing and memory usage during run-time
- but at the cost of more work to be done during compilation

When used to declare variables, these are implicitly **const**s. Example:

```
const int val1 = 3;           // evaluated at compile-time
const int val2 = val1 * 2;    // evaluated at compile-time
int a = 3;                    // variable a is not constant
const int val3 = a;           // evaluated at run-time
constexpr int c1 = val1;      // evaluated at compile-time
// constexpr int c2 = val3; // compile error, val3 is not constant
```

14.2. **constexpr** – Generalized constant expressions

constexpr can precede a function or method. In that case it will be evaluated at compile-time only when all the arguments are evaluated at compile-time:

```
constexpr int square(int value) {
  return value * value;
}
square(4);  // evaluated at compile-time (4 is const)
int val = 4;
square(val);  // evaluated at run-time (val is non-const)
```

If a function has run-time features (e.g., try-catch, assertions*, virtual**, static***, non- constexpr functions, et .), it will be evaluated at run-time.
[*: allowed since C++14 (*), C++20 (**), C++23 (***)]

14.2. **constexpr** – Generalized constant expressions

constexpr non-static class methods of run-time objects cannot be used at compile-time if they contain data members or non-compile-time functions:

```
struct Value {
   int val = 3;
   constexpr int getVal() const { return val; }
   static constexpr int get3() { return 3; }
};
```

```
Value v1;
constexpr Value v2;
// constexpr int x = v1.getVal();  // compile error, method not constexpr
constexpr int y = v1.get3();  // same as 'Value::get3()'
constexpr int z = v2.getVal(); // works
```

14.2. **constexpr** – Generalized constant expressions

Since C++17, **if constexpr** can be used to compile code on a condition:

```
auto myVersion() {
  if constexpr (__cplusplus == 202101L)  // __cplusplus macro holds c++ version
    return "C++23";   // const char*
  else
    return 11;  // int, returned when c++ version is not 20
}
```

Since C++20, two more keywords can be used:

- **consteval** guarantees compile-time evaluation and will produce an error when run-time arguments are supplied
- **constinit** guarantees compile-time initialization of variables and will produce an error when run-time arguments are supplied. This is weaker than **constexpr**, since the initialized variable *can* change its value later.

14.3. Move semantics

In C++, ***the rule of three*** is a guideline, which states that if a class defines any of the following three, then it should explicitly define all three:
(1) destructor, (2) copy constructor, and (3) copy assignment operator
to avoid their default implementation during compilation (which is usually incorrect).

Since C++11, ***the rule of five*** expands this for these two additional special *move semantics* methods:
(4) move constructor, and (5) move assignment operator
for the same reason.

More details: https://en.cppreference.com/w/cpp/language/rule_of_three

14.3. Move semantics

Lvalue (Left Value) is something that has a name and a memory address. It can appear on the left-hand side of an assignment and you can take its address with &.

```
int x = 10;     // x is an lvalue
x = 20;         // valid: lvalue on left side
int * p = &x;   // valid: you can take the address of x
```

Rvalue (Right Value) is a temporary value that doesn't have a name or address. It can appear only on the right-hand side of an assignment and you can't take its address directly.

```
int x = 10;
x = 10 + 20;      // 10 + 20 is an rvalue
int y = x * 2;    // x * 2 is an rvalue
//int * p = &(x + 1);  // Error: can't take address of rvalue
```

14.3. Move semantics

Since C++11:

Rvalue references (`Classname &&`) let you bind to rvalues (see move constructors). `Classname &` binds to lvalues (named objects), `Classname &&` binds to rvalues (temporary objects). Rvalue references let you "steal" resources from temporary objects, rather than copying them. This is the foundation of move semantics.

`std::move()` can be used to convert lvalues into rvalues intentionally

```
std::string && strRef = std::string("Hello");  // rvalue reference
                          // to temporary string, strRef is now "Hello"
strRef += ", world";      // strRef can be modified, is now "Hello, world"
std::string s = "hi";     // s is lvalue
// std::string && badRef = s;  // Error: binding rvalue reference to lvalue
```

14.3. Move semantics -- rule of three

Example: Message class

Message_v1.cpp

```
class Message {
   char * text;
  public:
   Message(const char * str);                  // constructor with C string
   ~Message();                                 // 1. Destructor
   Message(const Message & other);             // 2. Copy constructor
   Message & operator=(Message & other);       // 3. Assignment operator

   // friend method that returns a reference to a concatenated string:
   friend Message & operator+(const Message & m1, const Message & m2);

   void show() { std::cout << text << '\n'; }
};
```

14.3. Move semantics -- rule of three

Example: Message class

Message_v1.cpp

```
Message::Message(const char * str) {   // copy from str:
  text = new char[std::strlen(str) + 1];
  std::strcpy(text, str);
}

Message::~Message() { delete[] text; }

Message::Message(const Message & other) {  // perform deep copy:
  text = new char[std::strlen(other.text) + 1];
  std::strcpy(text, other.text);
}

Message & Message::operator=(Message & other) {
  std::swap(*this, other);  // see copy-swap idiom
  return *this;
}
```

14.3. Move semantics -- rule of three

Example: Message class

Message_v1.cpp

```
// friend operator that concatenates two Messages:
Message & operator+(const Message & m1, const Message & m2) {
  char * text = new char[std::strlen(m1.text) + std::strlen(m2.text) + 1];
  std::strcpy(text, m1.text);
  std::strcat(text, m2.text);
  Message result(text);
  delete[] text;
  return result;  // return by value or move
}
```

14.3. Move semantics -- rule of three

Example: Message class

Message_v1.cpp

```
int main() {
  Message s1("ping!");  // s1 is an object from C string
  Message s2(s1);       // s2's copy constructor from s1: lvalue
  Message s3(s1+s2);    // s3's copy constructor from s1+s2: rvalue?
  Message s4 = s1;      // s4's assignment operator copies
  s1.show(); s2.show(); s3.show(); s4.show();
}
```

In the above, **`s1`** as a parameter to s2's copy constructor is an **lvalue**.

`(s1+s2)` as a parameter for s3's copy constructor *could* be an **rvalue**, a temporary object that is removed after the statement on that line is finished.

If it were an **`rvalue`**, the move constructor would be called instead of the copy constructor, allowing for better performance: see next slides.

14.3. Move semantics -- rule of five

Example: Message class, *with* rule of five

Message_v2.cpp

```
class OwnString {
   char * text;
 public:
   Message(const char * str);                    // single constructor with C string
   ~Message();                                   // 1. Destructor
   Message(const Message & other);               // 2. Copy constructor
   Message & operator=(const Message & other);   // 3. Assignment operator
   Message(Message && other);                    // 4. Move constructor
   Message & operator=(Message && other); // 5. Move assignment operator
   // friend method that returns a Message having a concatenated string:
   friend Message operator+(const Message & m1, const Message & m2);
   // print out the Message object address and text:
   void show() { std::cout << this << ':' << (text?text:"?") << '\n'; }
};
```

14.3. Move semantics -- rule of five

Example: Message class, *with* rule of five

OwnString_v2.cpp

```
int main() {
  Message s1("ping!");  // s1 is an object constructed from C string
  Message s2(s1);       // s2's copy constructor from s1: lvalue
  Message s3(s1+s2);    // s3's constructor from s1+s2: rvalue
  Message s4 = s1;      // s4 is copy-constructed here by the compiler
  Message s5("pong!");  // s5 is constructed here
  s5 = s2;              // s5 is assigned here
  Message s6 = std::move(s2);   // s6 is move constructed here
  s1.show(); s2.show(); s3.show(); s4.show(); s5.show(); s6.show();
}
```

14.4. Measuring Time

For measuring how long a program needed to perform a task, there are three types of time measurement:

- **Wall-Clock/Real time**: Human-perceived passage of time from the start to the completion of a task (includes other processes taking resources, too)
- **User/CPU time**: The time spent by the CPU to process user code
- **System time**: The time spent by the CPU to process system calls (including I/O calls) executed into kernel code

14.4. Measuring Time - Wall-clock time

On Linux / MacOSX (resolution in microseconds):

WallClock.cpp

```
#include <time.h>        //struct timeval
#include <sys/time.h>    //gettimeofday()
#include <iostream>

int main() {
  struct timeval start, end;  // struct timeval {second, microseconds}
  ::gettimeofday(&start, NULL);
  double ret = 0;
  for (int i=0; i<0xFFFFF; i++) { ret += ret*0.3; }  // task to be measured
  ::gettimeofday(&end, NULL);
  long start_time = start.tv_sec * 1000000 + start.tv_usec;
  long end_time = end.tv_sec * 1000000 + end.tv_usec;
  std::cout << "Time: " << end_time - start_time << " microsecs.\n";
}
```

14.4. Measuring Time - User time

Using **`std::clock`** (resolution in nanoseconds):

UserTime.cpp

```
#include <chrono>  // clock_t, std::clock
#include <iostream>

int main() {
  clock_t start_time = std::clock();
  double ret = 0;
  for (int i=0; i<0xFFFFF; i++) { ret += ret*0.3; }  // task to be measured
  clock_t end_time = std::clock();
  float diff = static_cast<float>(end_time - start_time);  // static_cast
  diff /= CLOCKS_PER_SEC;  // POSIX-defined as 1000000
  std::cout << "Time: " << 1000*diff << " milliseconds \n";
}
```

14.4. Measuring Time - User & System time

Using <sys/times.h> (resolution in milliseconds):

UserSystemTime.cpp

```
#include <unistd.h>   // _SC_CLK_TCLK
#include <sys/times.h>  // struct ::tms
#include <iostream>

int main() {
  double ret = 0;
  struct ::tms start_time, end_time;
  ::times(&start_time);
  for (long i=0; i<0xFFFFFFFF; i++) { ret += ret*0.3; }  // task to measure
  ::times(&end_time);
  auto user_diff = end_time.tms_utime - start_time.tms_utime;
  auto sys_diff = end_time.tms_stime - start_time.tms_stime;
  float user = static_cast<float>(user_diff) / ::sysconf(_SC_CLK_TCK);
  float system = static_cast<float>(sys_diff) / ::sysconf(_SC_CLK_TCK);
  std::cout << "User Time:   " << user << " seconds \n";
  std::cout << "System Time: " << system << " seconds \n";
}
```