14. Performance

14.1. `const` correctness

14.2. `constexpr` – Generalized constant expressions

14.3. Move semantics

14.4. Measuring Time

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
```

For the above to-const parameter functions, the C++ compiler checks whether the passed is 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. A non-const variant and the const one for an object/variable can be thought of as different types.

const overloading of methods or operators allows const correctness:

```
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.
  return 0;
}
```

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
int val = 4;
square(val);  // evaluated at run-time
```

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:

```
class A {
 public:
  int v = 3;
  constexpr int f() const { return v; }
  static constexpr int g() { return 3; }
};
```

```
A a1;
// constexpr int x = a1.f();  // compile error, f() not constexpr
constexpr int y = a1.g(); // works, same as 'A::g()' since g() is static
constexpr A a2;
constexpr int z = a2.f(); // works
```

14.2. **constexpr** – Generalized constant expressions

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

```
auto f() {
  if constexpr (__cplusplus == 202101L)  // __cplusplus macro holds c++ version
    return "C++23";   // const char*
  else
    return 3;  // 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

Example: OwnString class

OwnString_v1.cpp

```
class OwnString {
 public:
  OwnString(const char * p);                     // constructor with C string

  ~OwnString();                                  // 1. Destructor
  OwnString(const OwnString & that);             // 2. Copy constructor
  OwnString & operator=(OwnString & that);// 3. Assignment operator

  // friend method that returns a reference to a concatenated string:
  friend OwnString & operator+(const OwnString & s1, const OwnString & s2);
  void show() { std::cout << data << '\n'; }

 private:
  char * data;
};
```

14.3. Move semantics

Example: OwnString class

OwnString_v1.cpp

```
OwnString::OwnString(const char * p) {
  size_t size = std::strlen(p) + 1;
  data = new char[size];
  std::memcpy(data, p, size);
}
OwnString::~OwnString() { delete[] data; }
OwnString::OwnString(const OwnString & that) {
  size_t size = std::strlen(that.data) + 1;
  data = new char[size];
  std::memcpy(data, that.data, size);  // deep copy of that.data to data
}
OwnString & OwnString::operator=(OwnString & that) {
  std::swap(data, that.data);  // see copy-swap idiom
  return *this;
}
```

14.3. Move semantics

Example: OwnString class

OwnString_v1.cpp

```
// friend operator:
OwnString & operator+(const OwnString & s1, const OwnString & s2) {
   size_t size = std::strlen(s1.data) + std::strlen(s2.data) + 1;
   char * data = new char[size];
   std::memcpy(data, s1.data, std::strlen(s1.data));
   std::memcpy(data+std::strlen(s1.data), s2.data, std::strlen(s2.data));
   OwnString * s = new OwnString(data);
   return * s;
}
```

14.3. Move semantics

Example: OwnString class

OwnString_v1.cpp

```
int main() {
  OwnString s1("ping!");  // s1 is an object from C string
  OwnString s2(s1);       // s2's copy constructor from s1: lvalue
  OwnString s3(s1+s2);    // s3's copy constructor from s1+s2: rvalue?
  s1.show(); s2.show(); s3.show();
  return 0;
}
```

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

Example: OwnString class, *with* move constructor: Note the differences

OwnString_v2.cpp

```
class OwnString {
 public:
  OwnString(const char * p);                  // constructor with C string

  ~OwnString();                               // 1. Destructor
  OwnString(const OwnString & that);          // 2. Copy constructor
  OwnString & operator=(OwnString that);      // 3. Assignment operator (no ref)
  OwnString(OwnString&& that);                // move constructor: OwnString&&
                                              //        is an rvalue reference
  // friend method that returns an rvalue :
  friend OwnString && operator+(const OwnString & s1, const OwnString & s2);
  void show() { std::cout << data << '\n'; }

 private:
  char * data;
};
```

14.3. Move semantics

Example: OwnString class

OwnString_v2.cpp

```
// copy constructor:
OwnString::OwnString(const OwnString & that) {
  size_t size = std::strlen(that.data) + 1;
  data = new char[size];                  // deep copy of that.data to data
  std::memcpy(data, that.data, size);     // lots of work
}


// move constructor:
OwnString::OwnString(OwnString&& that) {  // OwnString&& is an rvalue
  data = that.data;                       // reference to a string
  that.data = nullptr;    // note the simplicity (versus copy constructor)
}
```

14.3. Move semantics

Example: OwnString class, *with* move constructor

OwnString_v2.cpp

```
// friend operator returning an rvalue:
OwnString && operator+(const OwnString & s1, const OwnString & s2) {
  size_t size = std::strlen(s1.data) + std::strlen(s2.data) + 1;
  char * data = new char[size];
  std::memcpy(data, s1.data, std::strlen(s1.data));
  std::memcpy(data+std::strlen(s1.data), s2.data, std::strlen(s2.data));
  OwnString * s = new OwnString(data);
  return std::move(* s);  // std::move will set to rvalue
}
```

14.3. Move semantics

Example: OwnString class, *with* move constructor

OwnString_v2.cpp

```
int main() {
  OwnString s1("ping!");  // s1 is an object from C string
  OwnString s2(s1);       // s2's copy constructor from s1 (an lvalue)
  OwnString s3(s1+s2);    // s3's move constructor from s1+s2 (an rvalue)
  OwnString s4=s3+s1;     // s4's assignment operator on move constructor
                          //  of s3+s1 (an rvalue)
  s1.show(); s2.show(); s3.show(); s4.show();
  return 0;
}
```

In the above, **s1** as a parameter to s2's copy constructor is an **lvalue**, whereas **(s1+s2)** as a parameter to s3's move constructor is an **rvalue**, a temporary object that is removed after the move constructor is finished.

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";
  return 0;
}
```

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";
  return 0;
}
```

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";
  return 0;
}
```