Modern C++

Modern C++

We call “modern” C++ to all features C++11 / C++14 / C++17 / C++20

- Yes, each version follow the year it is approved
- Remember that video game programming is not very fond to modern techniques
- Still, performance matters, and elements like the new move operator helps
    - Some help make things easier to read and stronger typed like nullptr or override
- Read Visual Studio compiler language conformance

lvalues and rvalues

- Values we are used to work with are **lvalues** (you can get an address to)
- Temporary objects are **rvalues**
- We can have const references of lvalues and rvalues
- Until C++11 we can only have (non const) references of lvalues

```
void inc(int& a)
{
    a = a + 1;
}

int inc2(const int& a)
{
    return a+1;
}

void main()
{
    int i = 0;
    inc(i);  // compilation ok
    inc(4);  // compilation error
    inc2(4); // compilation ok
}
```

rvalue references

- C++ 11 introduces the concept of rvalue reference
- It will be used to receive objects that we know are temporary (no longer be used), so we can do **optimizations**

```
class A
{
  int* ptr = nullptr;

  A(int* p) : ptr(p)) {}
  ~A() { delete p; }

  void copy(const A& a) // unoptimized lvalue ref
  {
    delete ptr;
    ptr = new int(*a.ptr);
  }

  void copy(A&& a)  // optimized rvalue ref
  {
    ptr = a.ptr;  // a is no longer be used
  }
};

void main()
{
    A a(new int(4));
    A b(new int(5));

    a.copy(b);               // unoptimized
    a.copy(A(new int(8));  // optimized version
}
```

rvalue references

- What if I want to use an rvalue reference from a lvalue?
    - **std::move**
- What if I want to pass an lvalue from a rvalue reference?
    - **std::forward**

```
class A
{
  int* ptr = nullptr;

  A(int* p) : ptr(p)) {}

  void copy(const A& a) // unoptimized lvalue ref
  {
    delete ptr;
    ptr = new int(*a.ptr);
  }

  void copy(A&& a)  // optimized rvalue ref
  {
    ptr = a.ptr;  // a is no longer be used
  }

  void copy2(A&& a)
  {
    copy(std::forward<const A&>(a)); // calls unoptimized
  }
};

void main()
{
    A a(new int(4));
    A b(new int(5));

    a.copy(std::move(b));  // calls optimized
    a.copy2(std::move(b));
}
```

Move constructor and assignment operator

- rvalue references comes with new **move constructor and assignment operator**
- Both are used to optimize copy constructor and assignment operator for temporary objects (rvalue references)

```
class A
{
  int* ptr = nullptr;

  A(int* p) : ptr(p) {}
  ~A() { delete ptr; }
  A(const A& a) : ptr(new int(*a.ptr)) {}
  A(A && a) : ptr(a.ptr) { a.ptr = nullptr; }

  A& operator=(const A& a)
  {
     delete ptr;  new int(*a.ptr);
     return *this;
  }

  A& operator=(A && a)
  {
     delete ptr; ptr = a.ptr; a.ptr = nullptr;
     return *this;
  }
};

void main()
{
    A a(new int(4));
    A b(a);               // copy constructor
    A c(std::move(a));    // move constructor
    A = b;                // assignment
    A = std::move(b);     // move assignment
}
```

Optimizations: containers

- rvalue references optimizes stl containers like **std::vector**.
- Without rvalue references, inserting into a std::vector could lead to **copy constructor** calls.
- With rvalue references **move constructors** are called instead.

```
void main()
{
    std::vector<std::string> names;

    names.push_back(std::string("carlos"));

    // this line calls copy/move constructor for
    // string "carlos".
    names.insert(names.begin(),
std::string("marc"));

    // calling copy constructor for std::string
means
    // a new allocation and copying each of the
chars
    // into new string

    // calling move constructor for std::string
means
    // copying a char* pointer.
}
```

Optimizations: return values and perfect forwarding

- With rvalue references temporary values created to hold return value of functions are removed (only when returning local variables)
- rvalue references allow perfect forwarding of parameters removing also temporary values and copies. See **emplace_back** function in std::vector

```
std::vector<int>  get_values()
{
  // without rvalue references vector must be
  // copied after return . Which means another
  // allocation, deallocation and copying 3
values

return std::vector<int>({1, 2, 3});
}

struct file
{
   std::string path, name;

   file(const std::string& _path,
        const std::string& _name)
            : path(_path), name(_name) {}
};

void main()
{
    std::vector<int> v = get_values();

    std::vector<file> names;
    names.emplace_back(std::string("/user/"),
                       std::string("log.txt"));

    // strings are forwarded to file constructor
    // without extra copies and allocations
}
```

Smart Pointers

Smart pointers are classes that automatically manages life of a given object:

- unique_ptr
  - Takes **ownership** of a pointer (removes it when destructor is called).
  - Only one unique_ptr can own the same pointer. Trying to **copy** a unique_ptr compilation will fail.
  - Implements move constructor and **move assignment operator**. Useful for changing the ownership of the pointer and for adding unique_ptr to stl containers without performance cost.

Smart Pointers

- shared_ptr
  - uses a shared **reference count** to take into account multiple owners.
  - when destructor is called reference count is decremented and if count is 0 pointer is deleted
  - adds extra cost of **allocating a new pointer** (the shared reference count)
  - each copy/destruction of a shared_ptr requires incrementing/decrementing de reference which means also an extra cost.

Smart Pointers

- weak_ptr
    - does NOT hold **ownership** of a pointer
    - is able to check if a pointer contained in a shared_ptr is still valid (not deallocated) If pointer is valid can be accessed.
    - Adds extra cost to shared_ptr when used
    - Are useful to break **circular reference counting** : two objects having a shared_ptr one to each other.

Constexpr

The concept is to have compile time constant values stored in a read-only memory. It is somehow like const but with real optimization impact:

```
constexpr int size = 10;
constexpr int test() { return 10; }

int a[size];
int b[test()];
```

Under specific circumstances, constexpr can be used in member / constructors

Auto

- auto is used to compile time deduction of types. This can lower the verbosity of our code (use it with caution)
- Remember that auto will ignore references and const
  - `auto a = (const int&) b; // a is of type int`
  - `const auto a& = (const int&) b; // a is of type const int&`
- As of C++14 you can actually return auto

```
auto add(int a, int b)
{
    return a+b;
}
```

```
for(auto it = vect.begin(); it != vect.end(); ++it)
{
    std::cin >> *it;
}
```

Decltype

- C++11 standardized typeof() to decltype() to have the compiler deduce the type of any expression at compile time (not to confuse with typeid() of RTTI)

```
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1;  // type is const int&&
decltype(i) x2;      // type is int
decltype(a->x) x3;   // type is double
decltype((a->x)) x4; // type is const double& (lvalue expression)
```

Lambda Expressions

C++11 defines lambdas for writing embedded code functions:

```
[capture clause] (params) -> return_type
{
   // definition
}
```

```
auto lambda = [peer](const std::string& name) → bool
{
    return strcmp(peer->username, name) == 0;
}

auto it = std::find_if(users.begin(), users.end(), lambda);
```

Lambda Expressions

- Return type is optional and most of the time deduced by the compiler
- Capture clause defines level of visibility of the lambda:
  - [ ] only local variables inside lambda are visible
  - [=] all symbols are accessible by value
  - [&] all previous symbols are accessible by reference
  - [a, &b] captures <u>a</u> by value and <u>b</u> by reference
  - [this] captures the <u>this</u> pointer

```
std::vector<int> c({1,2,3,4,5,6,7});
int x = 5;
c.erase(std::remove_if(c.begin(), c.end(),[x](int n) {return n<x;}), c.end())
```

Generalized Lambda Expressions from C++14

We can now use auto as an argument to lambdas, sort of like *templating* their types:

```
auto sum = [](auto a, auto b)
{
    return a+b;
}

// integers
std::cout << sum(1, 6) << endl;

// floats
std::cout << sum(1.0f, 5.6f) << std::endl;

// strings
std::cout << sum(std::string("geeks"), std::string("forgeeks")) << std::endl:
```

Template variables

- As of C++14 we can use templates to declare unspecified typed variables

```
template<typename T>
constexpr T pi = T(3.14159);

template<>
constexpr const char* pi<const char*> = “pi”;

float f = pi<float>;
const char* pi_name = pi<const char*>;
```

What about other IDEs ?

- Visual Studio keeps being the reference IDE in video game programming
- XCode is the traditional option for Mac
- Consider having no IDE or very light ones:
  - Good editor like Visual STudio Code, 4code, atom, sublime text, vim, emacs, etc.
  - Makefile to pick compiler options and files to compile, binded to a key
  - Debug with a Visual Studio

Other compilers than Microsoft's ?

Microsoft compiler is considered only of mid-quality compared to:

- GCC is free software and compiler of choice for unix (sony / nintendo) systems
- Clang is quickly replacing gcc as the compiler of choice for its excellence optimization techniques. It includes a very good **static code analysis**
- Intel compiler is traditionally very good in optimization but complex to use

Adding Static Code Analyzers

A fantastic way of improving your C++ is via a static code analyzer:

- Sonarlint
- HelixQAC
- Cppcheck (free)
- clang static analyzer (free)

Homework: String class

String class that could be used like this (you **cannot** use string.h):

```
String a(“hello”)

String b(a)

String c = a + b;
```

```
If (c == “hellohello”) …

c.length() // string length

c.clear() // set to empty string
```

Extra bonus: add needed code to string so that this call is efficient

```
String GetMeAString() { return String("another string"); }

void main() { String a = GetMeAString(); }
```