C++ Primer for CS175

Yuanchen Zhu

September 11, 2013

This primer is nine pages long and might scare you. Don't worry! To do the assignment you don't need to understand every point made here. However this document does explain some of the design rationals behind the codes we provide, so you can refer back here if you get confused.

1 General C++ Usage

- In addition to the `/*` and `*/`, double forward slashes (`//`) marks the rest of the line as comment.
- Variable declarations don't have to be at the beginning of the scope and can be inside the `()` of a `for` loop. The following is legitimate C++:

```
int main() {
  int a = 10, b = 20; // declared at start of function, like C

  f(a, b); // f and g are functions
  g(a, b);

  int d = a * b;  // declared after regular statements

  for (int i = 0; i < d; ++i) { // declared in for (...)
    int f = d * i; // declared inside a {...}
    do_something_with_int(f);
  }

  // the variable f is no longer accessible.

  return 0;
}
```

  In fact it is considered good style to declare a variable as close as possible to its places of usage.

- Headers
  - For headers in the C standard lib: Remove ".h" suffix and add "c" prefix. E.g., `#include <stdio.h>` becomes `#include <cstdio>`
  - C++ standard lib: No suffix/prefix. E.g., `#include <string>`
- C++ namespace: big container of types/functions/variables to reduce name conflicts.
  - Everything in the standard C++ lib is in the `std` namespace. Use `std::` in front of type/function names to reference stuff in the namespace. For example, to reference a standard C++ string type, use `std::string`;
  - To save typing, you can also include an entire namespace at once: `using namespace std;` So I won't type `std::` from now on.

- Formatted input/output: use special operators `<<` and `>>` instead of `printf`.
  - `printf("Hello World!");` becomes `cout << "Hello World!";`
  - Compared with printf, the `<<` and `>>` operators are type-safe. What's wrong (and dangerous) with the following?

    ```
    printf("%d %d %s %d %d", 1, 2, 3, "apple", 4);
    ```

- Typecast: C++ still lets you cast anything to anything, but you're encouraged to tag the kind of cast you're making (for potentially unsafe casts).

  - `static_cast, dynamic_cast, const_cast, reinterpret_cast`

  - See C++ FAQ at the end of this document for detailed explanations.

2 Class

Think of as a glorified `struct`.

- Can define member functions, which are like normal functions, but have an implicit parameter `this`.
- Can contain member variables.
- *Constructor*: functions with the same name as the class. Piece of code called whenever a new instance of the class is instantiated.

  E.g.: `return Cvec3(10.0, 10.0);`

- *Destructor*: piece of code called automatically whenever an instance of the class is deallocated or goes out of scope. If a class is named `A`, its destructor must be named `~A`. A destructor doesn't taken input, and doesn't return anything.

  Example:

```
class Foo {
   int a_;
public:
   Foo() { a_ = 0; cout << "Foo() get called\n"; }
   Foo(int a ) { a_ = a; cout << "Foo(" << a << ") get called\n"; }
   ~Foo() { cout << "~Foo() get called with a = " << a_; }

   int get() const {return a_; }
}

int main() {
  Foo f; // prints "Foo() get called"

  Foo g(20); // prints "Foo(10) get called"

  // do stuff using f and g
  int c = f.get() + g.get();
  cout << c; // prints "10"

  // When the function returns, f and g go out of scope
  // and their destructors get called.
  return 0; // prints "~Foo() get called with a = 20"
            // followed by "~Foo() get called with a = 0"
}
```

- Functions and variables can have different visibility: `public`, `private` (the default), `protected`.
- In C++, a `struct` is the same as a `class` except its default member visibility is `public`.
- Inside a member function, `this` variable points to the current object.

3 Operator overloading

- Operator is no different from functions, and can be customly defined in C++. This is useful for doing math on customly defined data types.
- Later in the class, you have access to `class Cvec3` and `class Matrix4`. Given two variables: `Cvec3 a, b;`, can write `a += b;` instead of, say, `a.add(b);`
- Operators you're probably going to use:
  - assignment: `a = b;`
  - arithmetic's: `a *= b; c = a + b;`
  - indexing: `Cvec3 a(10.0, 10.0, 10.0); a[0] += 10.0;` For a `Matrix4 m`, `m(i,j)` gives you the element on the i-th row, j-th column. Wherease `m[k]` gives the k-th element when `m` is interpreted as a row-major linear array. For a `Cvec`, `v[i]` and `v(i)` have the same meaning.

4 Reference

- Behind the scene: like pointer, but does not need `&` and `*`. Unlike a pointer, a reference should never be NULL.
- Originally, you write:

  ```
  void getX(int *x) { *x = 10; }
  getX(&i); // i == 10 now
  ```

  Technically it is valid syntax for the caller to call `getX(NULL)`. Shall we test it in the body of `getX`?
- But now: `void getX(int &x) { x = 10; }`
  `getX(i); // i == 10 now`

  When you use reference, you are signaling the caller that address passed in should reference a valid object. (Of course, with some gnarly pointer magic, you can still dereference a NULL ptr and pass it in...)
- Useful for passing large object by reference. Often in combination with `const` modifier to make sure object is not modified. Here is an example.

```
// We declare a huge data structure of 10M in size
const static int N = 1024 * 1024 * 10;
struct Huge {
  unsigned char data[N];
};

// Pass Huge by value. Each invocation of sum_by_val incur the cost of
// copying 10M of data
int sum_by_val(Huge h) {
  int x = 0;
  for (int i = 0; i < N; ++i)
    x += h.data[i];
```

```
  // we can write to h, but this side-effect won't be observed by the caller since
  // we are working on a copy of the Huge passed in by the caller.
  h.data[0] = 123;
  return x;
}

// Pass by reference. Invocation of sum_by_ref incur no copying cost since only
// the address of the supplied argument is passed in.
int sum_by_ref(Huge& h) {
  int x = 0;
  for (int i = 0; i < N; ++i)
    x += h.data[i];

  // But now modifiying h resulting in the original argument being modified
  h.data[0] = 123;
  return x;
}

// Pass by const reference. Invocation of sum_by_cref incur no copying cost since only
// the address of the supplied argument is passed in. Moreover, the body of the function
// cannot modify h.
int sum_by_cref(const Huge& h) {
  int x = 0;
  for (int i = 0; i < N; ++i)
    x += h.data[i];

  // The following line would trigger a compilation error if uncommented
  //h.data[0] = 123;
  return x;
}

int main() {
  Huge h;
  sum_by_val(h); // this makes a temporary copy of 10M of data. h won't be modified.
  sum_by_ref(h); // this is fast, but h can be modified by the function call.
  sum_by_cref(h); // this is fast, and h is guaranteed not to be modified.
}
```

- Can be misused since you no longer know if the variable passed to getX will be modified or not, unless there's a `const` modifier. So the convention is: **ALWAYS** use a `const` modifier for input parameter that are passed by reference, and any passed-by-reference parameter without a `const` modifier is assumed to be modified by the function.

5 Templates

- Syntax example: `template <typename T, int n> class Cvec {...}`
- A form of smart macro expansion to avoid code duplication. E.g. `Cvec3` is actually an alias for the `Cvec` template above with parameter `T = double` and `n = 3`.
- Can be used essentially as a normal class.

6 Overloading

- Overloading = same name, multiple definition. But without confusing the compiler.
- Use different parameter list to differentiate.
- Constructors can be overloaded.

7 Standard Template Library (STL)

- Well thought out container library for C++
- Almost always better than rolling your own.
- `vector<int> a;` instead of `int a[100];`
- `string a("abc"), b("abc"), c = a + b;` instead of worrying that your `char[]` is of enough length.
- To replace a container of `Stuff` implemented as array with a container implemented as a double-linked list, simple write `list<Stuff>` instead of `vector<Stuff>`
- STL containers provide a standard `iterator` based interface to access their contents. Using this interface, STL provides multitudes of algorithm (such as binary search, quick sort, heap sort, etc) that works on all the containers.
- For our codes, we only use `vector` and `string`.

8 Dynamic allocation

- Use `A *p = new A` instead of `A *p = malloc(sizeof(A))` to allocate a single A, and then use `delete p` to free it.
- Use `A *pp = new A[10]` instead of `A *pp = malloc(sizeof(A) * 10)` to allocate 10 A's, and then use `delete[] pp` to release.
- Unlike `malloc` and `free`, `new` and `delete` will call the constructors and destructors after allocating the memory and before freeing the memory.
- Moreover, `new` will never return a NULL pointer. If it cannot allocate the required memory from heap, it will throw an exception (we will talk about exceptions later)
- In our codes, we use *Smart pointers* to keep track of dynamically allocated memory. The one we use is called `shared_ptr`. Example usage:

```
int f() {
  shared_ptr<A> p;

  // this makes p points to a newly allocated A;
  p.reset(new A);

  // work with p as if it is a normal pointer,
  do_stuff_with_A(*p);
  do_stuff_with_field_of_A(p->some_field);

  // don't need to call delete p.
  // As p goes out of scope the destructor of the
  // shared_ptr will automatically call delete
}
```

- Note that if you want to write `p.reset(new A[10])`, you should instead use a `vector<A> p(10)`. A `vector` is a dynamically allocated array of a specific type.
- Thus from above, you can see that in your code, **only** codes of the form `new A` can appear. Codes of the form `delete A`, `new A[10]`, `delete[] A`, `malloc`, `free` should **almost never** appear.

9 Exceptions

- Any object can be thrown as an exception, e.g., `throw 10;` or `string a("abc"); throw a`. The latter is the same as `throw string("abc")`.
- In provided codes, we mostly throw `runtime_error`, a standard exception defined in the header `<stdexcept>`.
- Exceptions are useful when they're thrown inside `try` blocks, e.g.,:

```
int main() {
  try {
    throw 100;

    cout << "This line of code will not be reached" << endl;
  }
  catch (int e) {
    cout << "An int exception " << e << " is caught";
  }
  catch (char e) {
    // will not be executed since 100 is not of type char.
    cout << "An char exception " << e << " is caught":
  }
}
```

- The exception, when thrown, cause normal "execution flow" to be interrupted. The flow of execution jumps directly to the end of the enclosing `try` block. Then the type of the object thrown (an `int` here) will be compared to the list of `catch` clauses. If there is a match, the body of statements following the corresponding catch clause is executed. If the exception is not caught by any `catch` clauses, it is re-thrown to the end of the next enclosing `try` block.
- More importantly, as the execution flow changes, some variables are forced to go out of scope. C++ guarantees that these out-of-scope variables' destructors get called if they have previously been constructed fully.

10 Resource Acquisition Is Initialization

The above contract between exception and calling destructors provides a powerful way to manage resources (heap allocated memory, file handles, database handles, opengl handles, etc) known as Resource Acquisition is Initialization (RAII), which we use to manage GL objects. That's why you see the `GlProgram`, `GlTexture`, `GlBufferObject` in `glsupport.h`.

Consider the following pieces of lengthy C codes that tries to properly manage memory

```
int f() {
  int *a = NULL, *b = NULL, *c = NULL;

  a = malloc(10); // grab 10 bytes
  if (a == NULL)
```

```
    return -1;

  b = malloc(20); // grab 20 bytes
  if (b == NULL) {
    free(a);
    return -1;
  }

  if (some_condition_is_true) { // suppose we want to return early
    free(a);
    free(b);
    return 2;
  }

  c = malloc(30); // grab 30 bytes
  if (c == NULL) {
    free(a);
    free(b);
    return -1;
  }

  // do work using a, b, and c;

  // free resources
  free(a);
  free(b);
  free(c);

  return 0;
}
```

Notice the repeated free's. A more clever way in C is to do

```
int f() {
  int *a = NULL, *b = NULL, *c = NULL, ret = -1;

  a = malloc(10); // grab 10 bytes from heap
  if (a == NULL)
    goto Cleanup;

  b = malloc(20); // grab 20 bytes
  if (b == NULL);
    goto Cleanup;

  if (some_condition_is_true) { // suppose we want to return early
    ret = 2;
    goto Cleanup;
  }

  c = malloc(30); // grab 20 bytes
  if (c == NULL)
    goto Cleanup;


  ret = 0;
```

```
Cleanup:
  if (a != NULL)
    free(a);
  if (b != NULL)
    free(b);
  if (c != NULL)
    free(c);
  return ret;
}
```

Now here is a the C++ version using RAII:

```
class Stuff {
public:
   char *p;
   Stuff(int n) {
     p = new char[n]; // if 'new' cannot allocate memory, it will throw an exception
   }
   ~Stuff() { delete[] p; }
}

int f() {
  // The following line invokes a's constructor, which allocates memory.
  // If this line finishes without throwing, a.p must points to the allocated memory.
  // If this line throws, execution of f is stopped and we jump to the end of
  // enclosing try block.
  Stuff a(10);

  // do stuff with a;

  // The following line invokes b's constructor, which allocates memory.
  // If this line finishes without throwing, b.p must points to the allocated memory.
  // If this line throws, execution of f is stopped and we jump to the end of
  // enclosing try block. Moreover, since a is fully constructed at this point, before
  // the jump is made, a's destructor is called, which frees the memory allocated by a
  Stuff b(20);

  // do stuff with a and b;

  if (some_condition_is_true) {
    // At this point, only a and b are fully constructed, so their destructors get called
    // when the function returns in the following line.
    return 2;
  }

  // Do some more stuff. If an exception is thrown somewhere, a and b will get destructed
  // automatically.

  // The following line invokes c's constructor, which allocates memory.
  // If this line finishes without throwing, c.p must points to the allocated memory.
  // If this line throws, execution of f is stopped and we jump to the end of
  // enclosing try block. Moreover, since both a and b are fully constructed at this point,
  // before the jump is made, a and b's destructors get called, which free the memory
  // allocated by a and b
```

```
  Stuff c(30);

  // do stuff with a, b, and c

  // When the function exits, a, b, and c go out of scope and their destructor gets
  // called, freeing the allocated memory
  return 0;
}
```

The above is of course a bit contrived. When the resource in question is heap memory, some predefined STL container or smart pointer usually suffices. Thus, in practice, people will just write:

```
int f() {
  vector<char> a(10);
  // do stuff with a
  vector<char> b(20);
  // do stuff with a and b
  vector<char> c(30);
  // do stuff with a and c
  return 0;
}
```

11 Other stuff

- OO ideas: encapsulation, inheritance, polymorphism, ... you probably already know from Java
- iostreams: a bit too complex and you're only going to use the `<<` and `>>` anyway.
- `const` correctness: we maintain it in the codes but you probably don't need to care.

12 Good C++ References

Again the goal of this course is not to train you into a C++ ninja, but just in case you're interested...

- "Think in C++" by Bruce Eckel. Free ebook at `http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`
- "C++ FAQ" at `http://www.parashift.com/c++-faq-lite/`.
- "Exceptional C++" and "More Exceptional C++" by Herb Sutter. You can grab them at Cabot Library.