Advanced C++

January 16, 2020

Mike Spertus

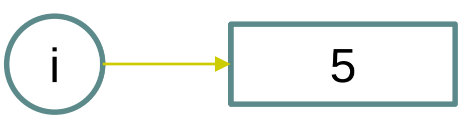
mike@spertus.com

The relation between code and data

- A program refers to data objects through expressions in C++ (or some other language)
- When the program is run, the actual data objects exist in computer memory
- How does the code find, interpret, and manage the objects at runtime?

Example: Simple variables

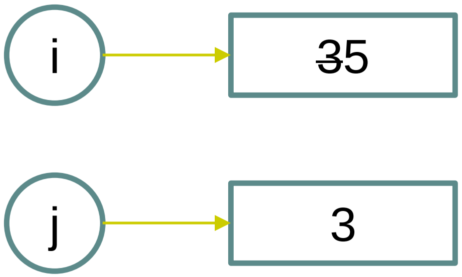
- Consider the following simple code
  - `int i = 5;`
- When the program creates the variable `i` at runtime, it allocates some memory (usually on the stack) to hold `i`'s data



- Any expressions in the code involving i will use the data there

Copying and assignment

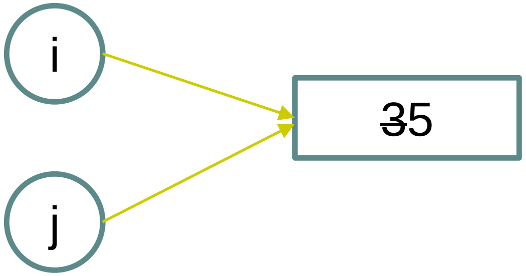
- In C++, when we do an assignment, the object is copied
- ```
  int i = 3;
  int j = i;
  i = 5;
  cout << j; // Prints 3
  ```



- https://godbolt.org/z/_GYLWx
- In this example, both i and j have their own storage associated with them in the running program
- **Warning:** In Java and Python, built-in types like int are copied, but user-defined types are not copied by assignment but are instead shared
  - cf next slide
  ```

References

- Suppose we wanted j to refer to the same data in memory as i, instead of a copy
- We can do that by creating j as a *reference* to an existing object (int &)
-
```
int i = 3;
int &j = i;
i = 5;
cout << j; // Prints 5
```



- https://godbolt.org/z/z2GDSr

References vs values

- The only difference between references and values is that the value has its own copy of the data, and the reference reuses the object it is initialized with as its storage without making a copy
- In the example on the previous slide
  - `i` is a variable of type `int` and has its own storage
  - `j` is of type `int&` ("int reference"), and does not have any new storage associated with it. It just treats i's storage as its own
- Both i and j can be used the same way after creation
  - If you are familiar with the use of & as the "address of" operator, this is an entirely different usage

Warning: Object lifetime

- If multiple variables are referring to the same object, you have to be careful that the object still exists when you use it
-
```
int &f() // f is a function returning an int&
{ int i = 3; return i; }

int &j = f(); // f returns a reference to f's i,
              // so the int& j also uses that as
              // its storage
// However, the storage for f's i is released when
// f is done running, so j has bound itself to a
// no longer existing object!
cout << j;    // Undefined behavior!
```

Dynamic memory management

- As with C, C++ programmers have to manage the lifetime of objects explicitly
  - In contrast to garbage collected languages like Java
- If memory is released too early, the program could crash from trying to use an object that no longer exists in the computer's memory
- If memory is not released when it is no longer needed, the program can run out of memory (a memory leak)

“Automatic manual memory management”

- In contrast to C, where memory management is very time consuming and error-prone, C++ has lightweight abstractions that, when used correctly, will automatically correctly managed the lifetimes of the object in memory
- The most common of these abstractions is unique_ptr

Creating a unique_ptr

- Calling make_unique<T> creates an object in memory on demand and returns a kind of handle to the object of type unique_ptr<T> that can be used to reference the object
- ```
  // Create an int in memory and return
  // an associated unique_ptr
  unique_ptr<int> ui = make_unique<int>(5);
  ```
  ```

A unique_ptr

- Gives you access to the data in the object
  - When you need a reference to the object managed by the unique_ptr, use the * operator
  - ```
    unique_ptr<int> ui = make_unique<int>(5);
    cout << *ui; // Prints 5
    ```
- Manages the lifetime of the object
  - When the unique_ptr goes away, it will automatically free up the memory of the object that it is managing
    ```

Updating a unique_ptr

- If you change a unique_ptr to point to a new object, it will free up the old one before it starts to manage the new one
- ```
  unique_ptr<int> ui = make_unique<int>(5);
  cout << *ui; // Prints 5
  ui = make_unique<int>(2);
  cout << *ui; // prints 2
  ```
- The unique_ptr automatically releases the memory of the first object (the one with value 5) before it starts managing the new object (the one with value 2)
  ```

Transfering ownership

- In our example program later today, we will need to transfer ownership from one unique_ptr to another
- This is a little tricky
- Assignment doesn’t work
  - up2 = up1; // Oops! Two “unique” owners
- Correct solution. Move from up1
  - up2 = move(up1); // up2 is owner. Don’t use up1
- We will learn more about moving soon
- But for now, you can treat it as a magic word

CLASSES

Creating our own types

- If we could only use a language' built-in types and couldn't define any of our own, the power of the language would be very much restricted to what was built-in
- In my solution to the Pascal's triangle problem, I sort of created my own Row and Triangle types
  - `typedef vector<int> Row;`
  - But what `typedef` does is let you give a new name to an existing type, so this is useful, but doesn't create genuinely new types

Classes

- To create your own type in C++, you must define a class
- Consider the following example (p. 61 in Koenig and Moo)

```
struct Student_info {
  string name;
  double midterm, final;
  vector<double> homework;
};  // Semicolon is required!
```

- See the sample “Grading” programs on Canvas
- Unlike C, no need for:

```
typedef struct Student_info Student_info;
```

Hey, that's a `struct`, not a `class`!

- That's OK, the only difference between a struct and a class in C++ is different default visibility of members.
- The following is equivalent

```
class Student_info {
public:
  string name;
  double midterm, final;
  vector<double> homework;
};
```

Visibility of members

- public members are visible to everyone
- Protected members are visible to subclasses
- Private members are only visible to the class

Visibility (cont)

```
class A {
 void f() {
    cout << pub; // OK
    cout << prot; // OK
    cout << priv; // OK
  }
public:
  int pub;
protected:
  int prot;
private:
  int priv;
};
class B : public A {
  void g() {
    cout << pub; // OK
    cout << prot; // OK
    cout << priv; // Error
  }
};

void h(A a)
{
    cout << a.pub; // OK
    cout << a.prot; // Error
    cout << a.priv; // Error
}
```

A class can have methods as well as fields

- You can also add member functions (methods) to go along with the data members (fields)

```
struct Student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  // Method to calculate the student’s grade
  double grade() const {
    return (midterm + final + median(homework))/3;
  }
};
```

Another way to organize the code

- In the previous slide, the code for how to calculate the grade was put right inside the class definition
- It is also possible to put it in a separate file (to avoid cluttering the interface)

```
// In .h file
struct Student_info {
  string name;
  double midterm, final;
  vector<double> homework;
  double grade() const;
};
// In .cpp file
double Student_info::grade() const
{
  return (midterm + final + median(homework))/3;
}
```

Accessing methods and fields

- Use the . operator
-
```
Student_info s;
s.name = "Mike";
s.midterm = 70;
s.final = 85;
s.homework.push_back(60);
s.homework.push_back(75);
cout << s.grade();
```

Static vs Dynamic types

- As we mentioned above, a program uses expressions to refer to objects in memory
- The static type is the type of the expression
  - Known at compile time
- The dynamic type is the type of the actual object referred to by the expression
  - May be knowable only at run-time
- Static and dynamic type generally only differ due to inheritance

(Single) inheritance

- One can use inheritance to model an “isA” relationship
- struct Animal { /* ... */ };
- class Gorilla : public Animal {…};
- This means that a Gorilla “isA” Animal and can be referred to by Animal references

Static type vs Dynamic type

```
int i = 5; // S = int, D = int
Gorilla g; // S = Gorilla, D = Gorilla
Animal &a = g; // S = An, D = Gor
Animal a2 = g; // Oops! Can't copy a Gorilla
               // into an Animal
unique_ptr<Animal> ua = make_unique<Gorilla>();
// Static type of *ua is Animal but Dynamic is Gorilla
ua = make_unique<Falcon>();
// Now S = Animal, D = Falcon
```

Virtual vs. non-virtual method

- A virtual method uses the dynamic type
- A non-virtual method uses the static type

Virtual vs. Non-virtual method

```
struct Animal {
  void f() { cout << "animal f"; }
  virtual void g()  { cout << "animal g"; }
};
struct Gorilla : public Animal{
  void f() { cout << "gorilla f"; }
  void g() { cout << "gorilla g"; }
   void h() { cout << "gorilla h"; }
};
void fun() {
   unique_ptr<Gorilla> g = make_unique<Gorilla>;
   Animal &a = *g;
   a.f(); // Not virtual: Animal’s f
   a.g(); // Virtual: Gorilla’s g
   a.h(); // Error: h is not in animal
   (*g).f(); (*g).g(); (*g).h(); // Gorilla’s f, g, and h
}
```

The -> operator

- In the previous slide, we used clunky expressions like `(*g).f()` to call the f method of the object managed by the unique_ptr g
  - `(*g)` is a reference to the Gorilla object managed by g
  - `(*g).f()` calls its f method
- This is so common that there is a special shortcut notation for it
  - `g->f(); // Same as (*g).f()`

Adding multiple grading strategies

```
struct Abstract_student_info { // In header
  string name;
  double midterm, final;
  vector<double> homework;

  istream& read(istream&);
  // Don’t define grading strategy here
  virtual double grade() const = 0;
};
```

Here are a couple of strategies

```
struct BalancedGrading: public Abstract_student_info {
  virtual double grade() const override {
    return (midterm + final + median(homework))/3;
  }
};

struct IgnoreHomework: public Abstract_student_info {
  double grade() const {
    return (midterm + final)/2; // Ignore the HW
  }
};
```

How do we choose which grading strategy to use?

```
int main()
{
  unique_ptr<Abstract_student_info> si
    = make_unique<Balanced_grading>();
  si->read(cin);
  cout << "Grade is " << si->grade() << endl;
  return 0;
}
```

Constructors

- `new Student_info()` leaves `midterm`, `final` with nonsense values. (Use the original version. The one with the "pure virtual" method can't be new'ed!)
- But not `homework`! We'll understand that momentarily
- Fix as follows:

```
struct Student_info {
  Student_info() : midterm(0), final(0) {}
};
```

Order of construction

- Virtual base classes first
  - Even if not immediate
- First base class constructors are run in the order they are declared
- Next, member constructors are run in the order of declaration
  - This is why we didn’t need to initialize homework. Vector’s constructor creates an empty vector
- This is defined, but very complicated
  - Best practice: Don’t rely on it
  - Good place for a reminder: Best practice: don’t use virtual functions in constructors

Constructor ordering

```
class A {
public:
  A(int i) : y(i++), x(i++) {}
  int x, y;
  int f() { return x*y*y; }
};
```

- What is `A(2).f()`?

Answer: 18! (x is initialized first, because it was declared first. Order in constructor initializer list doesn't matter)

Avoiding spaghetti inheritance

- While the examples above allow us to use different grading strategies, it quickly gets out of hand
- Suppose we also had classes like `MPCS_Student_info` and `Undergraduate_Student_info` inherit from `Abstract_Student_info`
- But then to cover all combinations, we would also need `MPCS_IgnoreHW_Student_info`, `MPCS_BalancedGrading_Student_info`, etc.
- The number of classes grows exponentially, and soon we have “spaghetti inheritance”
- The class `NewStudent_info` in the example program shows a way to make more focused use of inheritance to avoid this

Sample program

- All of these classes are demonstrated in the Grading programs on canvas
- You will need to compile the .cpp files together in a single project
- For example
  g++ -o Grading GradingTest.cpp Grading.cpp
- How to do this is compiler-specific, so you may need to check your documentation

HW 2.1 (Part 1)

- In addition to std::copy, there is a similar function in the <algorithm> header called std::transform, which lets you apply a function to each element before copying. Look up or Google for std::transform to understand the precise usage.
- Write a program that puts several doubles in a vector and then uses std::transform to produce a new vector with each element equal to the square of the corresponding element in the original vector and print the new vector (If you use ostream_iterator to print the new vector, you will likely get an extra comma at the end of the output. Don't worry if that happens).

HW 2.1 (Part 2)

- We will extend the program in part 1 to calculate and print the distance of a vector from the origin.
- There is also a function in the <numeric> header called `accumulate` that can be used to add up all the items in a collection.
  - (Googling for “accumulate numeric example” gives some good examples of using `accumulate`. We’re interested in the 3 argument version of `accumulate`).
- After squaring each element of the vector as in part 1, use accumulate to add up the squares and then take the square root. (That this is the distance from the origin is the famous Pythagorean theorem, which works in any number of dimensions).

HW 2.1 (Part 3)

- In real-life, you'd probably use `std::inner_product` to find the Euclidean length of a vector
- Learn about `inner_product` and use it to find a better way to accomplish part 2

HW 2.1 (Extra credit part 4)

- As yet another way to calculate the Euclidean length of a vector, is also a four argument version of `accumulate` that can combine `transform` and `accumulate` in a single step. Use this to provide another solution to part 2 of this problem

HW2.2

- One of the above slides referred to a function median, that takes the median of a vector of doubles.
- Part 1. Write the median function using the sort function in the algorithm header.
- Part 2. Write the median function using the `partial_sort` function in the algorithm header. Is this more efficient? Why do you think that? (You can give an intuitive or practical answer without precise mathematical analysis)
- Part 3. Write the median function using the `nth_element` function header. Do you think this is even more efficient? Why?
- Part 4 - Extra credit: Write a template function that can find the median of a vector of any appropriate type.
  - Although we haven't discussed writing our own template functions yet, looking at the template function for squaring from last week's slides
  - You can use any of the underlying algorithms from parts 1 to 3 above
- More extra credit. If there are an even number of elements, use the average of the middle 2 element

HW 2.3

- Rewrite the pascal.cpp file in Canvas (or your own) to be class-oriented