Object-oriented Programming

Lecture 4:
More on C++ and OOP in C++

Po-Chun Huang (黃柏鈞)

Agenda

- More on C++
- OOP in C++
  - Class
  - Function overloading

Pointers

- A **pointer** is a variable that contains the *memory address* of another variable, function, or object, called the **pointee**.

```
int* p; // pointer to an integer
int *q; // alternative syntax
int *a, *b, *c; // multiple pointers
int* a, b, c; // Only a is a pointer to int;
              // b and c are simply ints
```

Address-of operator (&)

- The address-of operator **&** returns the memory address of a variable:

```
int i = 10;
p = &i; // address of i assigned to p
```

Dereferencing Operator (*)

- The dereference operator (*) acting on a pointer returns the variable referred to by the pointer:

```
#include <iostream>
using namespace std;
int main() {
    int i = 10;
    int* p = &i;
    cout << "Address of i: " << p << endl;
    cout << "Value of i: " << *p << endl;
}
```

Working with Pointers

- When writing to the pointer, the same method is used. Without the asterisk, the pointer is assigned a new memory address, and with the asterisk the actual value of the variable pointed to will be updated.

```
p = &i;  // address of i assigned to p
*p = 20; // value of i changed through p
```

- If a second pointer is created and assigned the value of the first pointer, it will then get a copy of the first pointer’s memory address.

```
int* p2 = p; // copy of p
             // (copies the address stored in p)
```

Pointing to a Pointer

- Note that a pointer is also a variable, and can be retrieved with its memory address:

```
int** r = &p; // pointer to p (assigns address of p)
```

- Referencing the 2nd pointer now gives the address of the 1st pointer.
- Dereferencing the 2nd pointer gives the address of the variable.
- Dereferencing it again gives the value of the variable.

```
cout << "Address of p: " << r << endl; // ex. 0017FF28
cout << "Address of i: " << *r << endl; // ex. 0017FF1C
cout << "Value of i: "   << **r << endl; // 20
```

Dynamic Memory Allocation

- Declare **named variables** at compile time, allocate memory on **stack** ⇒ **static allocation**

```
int i;
```

- Allocate **anonymous memory** during runtime, allocate memory on **heap** ⇒ **dynamic allocation**

```
int* d = new int; // dynamic allocation
delete d; // release allocated memory
```

Null Pointer

- **Null pointers**: pointers not yet assigned to any valid addresses.
  - Classical way:

```
int* g = 0; // null pointer (unused pointer)
int* h = NULL; // null pointer
```

  - New way since C++11:

```
int* p = nullptr;
int i = nullptr; // error, but this is a good thing
bool b = (bool) nullptr; // false
```

Memory Access Violation

- Trying to dereference a pointer to released (invalid) memory space will cause a runtime error.

```
int* m = new int; // allocate memory for object
delete m; // deallocate memory
*m = 5; // error: memory access violation
int* n;
*n = 5; // error: memory access violation
```

- **Memory access violations are evil because they do not always crash your program immediately! Beware!**

Value of `nullptr`

- Deleted pointers should be set to null.
- Trying to delete an already deleted null pointer is safe. However, if the pointer has not been set to null, attempting to delete it again will cause memory corruption and possibly crash the program.

```
int* m = new int;
delete m;
m = nullptr; // mark as null pointer
delete m; // safe
```

Check Validity of Pointers

- Check the validity of a pointer before dereferencing it:

```
// check for valid pointer
if (m != nullptr) { *m = 5; }
if (m) { *m = 5; } // alternative
```

References

- **References** allow programmers to create new names (/**aliases**) for a variable, as a simpler and safer alternative to pointers that should be chosen whenever possible.
- **At the same time as the reference is declared, it must be initialized with a variable, and cannot be associated with another later.**

```
int x = 5;
int& r = x; // r is an alias to x
int &s = x; // alternative syntax
int& t; // error: must be initialized
r = 10; // assigns value to r/x
```

References vs. Pointers

- A **reference** is similar to a **pointer** that always points to the same thing, but…
  - A **pointer** is a variable that points to another variable.
  - A **reference** is only an alias and doesn’t have its own address.

```
int* ptr = &r; // ptr assigned address to x
```

References vs. Pointers (Cont’d)

- Generally, whenever a pointer does not need to be reassigned, a reference should be used instead, because a reference is safer.
- No need to check if a reference refers to null, as should be done with pointers.

```
int* ptr = nullptr; // null pointer
int& ref = *ptr;
ref = 10; // error: invalid memory access
```

Rvalue Reference

- The **rvalue reference** can *bind* and *modify* temporary objects (**rvalues**), such as literal values and function return values. An rvalue reference is formed by placing two ampersands **&&** after the type.

```
int&& ref = 1 + 2; // 3 is saved temporarily, not copied
```

- The rvalue reference extends the lifetime of the temporary object and allows it to be used like an ordinary variable.

```
ref += 3;
cout << ref; // 6
```

- **Rvalue references avoid unnecessary copying of temporary objects, thereby enhancing the run-time performance.**
  ⇒ Especially useful for large, complex objects.

Pass by Value (Cont'd)

```
#include <iostream>
#include <vector>
using namespace std;
void change(int i) { i = 10; }
void change(vector<int> a) { a.at(0) = 5; }
int main() {
   int x = 0; // value type
   change(x); // copy of x is passed
   cout << x; // "0"
   vector<int> v { 3 }; // reference type
   change(v); // object copy is passed
   cout << v.at(0); // "3"
}
```

Pass by Reference

- To instead **pass a variable by reference**, add an ampersand **&** before the parameter’s name in the function’s definition.
- With passed by reference, both primitive and object data types can be changed, which affects the original variable.

```
void change(int& i) { i = 10; }

int main() {
    int x = 0; // value type
    change(x); // reference is passed
    cout << x; // "10"
}
```

Pass by Address

- Arguments may also be **passed by address** using pointers. This passing technique serves the same purpose as passing by reference, but uses pointer syntax instead.

```
void change(int* i) { *i = 10; }
int main() {
  int x = 0; // value type
  change(&x); // address is passed
  cout << x; // 10
}
```

Pointer vs. Reference

- A pointer is a variable holding the main-memory address of another variable.
- A reference is simply an alias of another variable, and is bound to the same variable once initialized.
- Pointers can be null, whereas references cannot.
- If the function should not allow null arguments, it is preferable to use pass by reference to avoid **null pointer dereferencing**, which causes **memory access violation**.

Return by Value

- Normally, a function **returns by value**, in which case a copy of the value is returned to the caller.

```
int byVal(int i) { return i + 1; }
int main() {
  int a = 10;
  cout << byVal(a); // 11
}
```

Return by Reference

- To return by reference instead, place a ‘&’ after the return type.
  - The function must then return a nonlocal variable, not an expression or literal. This is different from returning by value.
  - Instead, return by reference is commonly used to return an argument that has also been passed to the function by reference.

```
int& byRef(int& i) { return i; }
int main() {
  int a = 10;
  cout << byRef(a); // "10"
}
```

Return by Address

- To return by address, append the dereference operator * to the function's return type.
- Likewise, the function must then return the address of a nonlocal variable, not that of an expression or literal.

```
int* byAdr(int* i) { return i; }
int main() {
  int a = 10;
  cout << *byAdr(&a); // "10"
}
```

Comparison

- It is unknown whether the caller (who calls the function) or callee (the called function) has allocated the memory space whose address is returned. (Dangerous!)
- Use **smart pointers** (see later) or **call by reference** instead.

Inline Functions

- Function calls causes small overheads on pushing the return address and parameters to the stack when called, and popping them from the stack when returned.
- Use **inline functions** to avoid this overheads by copying the function body directly to the called places of the function.

```
inline int myInc(int i) { return ++i; }
```

- Inline functions are merely a programmer's *recommendation*, not *mandatory*. Compilers may choose to inline a function or ignore the application.

Function Overloading

- **Function overloading** allows to implement the same operation on different but similar types, such as `int`, `long`, and even `float`.
- Example:

```
add(int a, int b);
add(double a, double b);
```

Example

```
#include <iostream>
using namespace std;
void add(int a, int b) {
  cout << "sum = " << (a + b);
}
void add(double a, double b) {
    cout << endl << "sum = " << (a + b);
}
// Driver code
int main() {
    add(10, 2);
    add(5.3, 6.2);
    return 0;
}
```

auto and decltype

- Both **auto** and **decltype** are for **type deduction** during compilation.
- While the auto keyword deduces the variable type by its initialization, the decltype keyword deduces and assigns the data type of a variable based on the parameter passed to it.
- auto is a placeholder of a type to instruct the compiler to try deduce the type by itself:

```
auto i = 5;      // int
auto d = 3.14;   // double
auto b = false;  // bool
```

https://www.scaler.com/topics/cpp/auto-in-cpp/

Auto Translates to Core Types Only

- The `auto` keyword translates to the core type of the initializer, which means that ***any reference and constant specifiers are dropped***:

```
const int& iRef = i;
auto myAuto = iRef; // int
```

- ***Dropped specifiers may be added back if needed***:

```
const auto& myRef = iRef; // const int&
```

Example

- What is the type of myAuto?

```
#include <iostream>
using namespace std;

int main() {
    int count = 10;
    int& countRef = count;
    auto myAuto = countRef;

    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;
}
```

https://learn.microsoft.com/en-us/cpp/cpp/auto-cpp?view=msvc-170

auto with && (Not R-value References)

- In the case of auto, && makes the compiler automatically deduce either an rvalue or an lvalue reference, based on the given initializer:

```
int i = 1;
auto&& a = i; // int& (lvalue reference)
auto&& b = 2; // int&& (rvalue reference)
```

auto Is Smart

```
#include <iostream>
#include <vector>
using namespace std;

// vector of ints
vector<int> myVector { 1, 2, 3 };

// "123"
for (auto& x : myVector) { cout << x; }
```

Advances in New Versions of C++

```
// C++03 and before
for(vector<int>::size_type i = 0;
    i != myVector.size();
    i++) {
cout << myVector[i]; // "123"
}

// C++11 and after can use this…
for (auto& x : myVector) { cout << x; } {
cout << myVector[i]; // "123"
}
```

Exercises

dedcuced

- What are the types of all underlined variables below?

最前面的 auto已經在初始化 m的時候帶入成 int這個型別了，所以後半會等同於 int &n = m ;

```
auto m = 1, &n = m;
auto x = 1, *y = &x, **z = &y;
auto a(2.01), *b (&a);
auto c = 'a', *d(&c);

int v1 = 100, v2 = 200;
auto v3 = v1 > v2 ? v1 : v2;
```

m int
n int 型別的參考（也就是別名）
x int
y int*
z int** int的指標的指標
a double
b double*
c char
d char* char的指標
v1 int
v2 int
v3 int

v1比v2大就return v1不然就return v2

https://learn.microsoft.com/en-us/cpp/cpp/auto-cpp?view=msvc-170

Exercises (Cont'd)

- What are the types of all underlined variables below?

```
int f(int x) { return x; }

int main() {
    auto x = f(0);
    const auto& y = f(1);
    int (*p)(int x); // Function pointer…
    p = f;
    auto fp = p;
}
```

https://learn.microsoft.com/en-us/cpp/cpp/auto-cpp?view=msvc-170

`decltype` vs. `auto`

- The **`decltype`** specifier works similar to **`auto`**, except that it deduces the exact declared type of a given expression, ***including references***:

```
int i = 1;
int& myRef = i;

auto a = myRef; // int
decltype(myRef) b = myRef; // int&
decltype(auto) c = myRef; // int&
```

The keyword `auto` is then replaced with the initializing expression, allowing the exact type of the initializer to be deduced.

`decltype` vs. `auto` (Cont’d)

- **`auto`** is simpler when an initializer is available. (Observable by the previous page’s example.)
- **`decltype`** is to forward function return types, without having to consider whether it is a *reference* or *value type*.

```
decltype(5) getFive() { return 5; } // int
```

Deducing the Types of Return Values

- The trailing return type syntax uses the type of a parameter as the type of the return value. (C++11 and later)

```
auto getValue(int x) -> decltype(x) { … } // int
```

The use of `auto` context just means that the *trailing return type* syntax is being used.

- Since C++14, the core return type can be deduced directly from the return statement. (C++14 and later)

```
auto getValue(…) { return x; } // int
decltype(auto) getRef(int& x) { return x; } // int&
```

Returning Multiple Values

- A convenient way to return multiple values from a function is to use a **tuple**:

```
#include <tuple>
#include <iostream>
using namespace std;
tuple<int, double, char> getTuple() {
  return tuple<int, double, char>(5, 1.2, 'b');
}
```

The Power of Auto

```
#include <tuple>
#include <iostream>
using namespace std;
tuple<int, double, char> getTuple() {
  return tuple<int, double, char>(5, 1.2, 'b');
}
auto getTuple2() {
  return make_tuple(5, 1.2, 'b');
}
```

Working with Tuples

- Individual tuple elements can be extracted with `std::get`.
  - Angle brackets (`<>`) are used to specify the index for the element to be retrieved.
  - The **type name** can be used to retrieve the element if there is only one element of that type.

```
auto getTuple() { return make_tuple(5, 1.2, 'b’); }
int main() {
  auto mytuple = getTuple();
  cout << get<0>(mytuple) // "5"
       << get<char>(mytuple); // "b"
}
```

Working with Tuples (Cont'd)

- Unpack a tuple with the **`std::tie`**, which copies one or more tuple elements to the provided arguments:

```
int main() {
  int i;
  double d;
  // Unpack tuple into variables
  tie(i, d, ignore) = getTuple();
  cout << i << " " << d; // "5 1.2"
}
```

Ignore the 3rd element of the tuple, not retrieving it.

Working with Tuples (Cont'd)

- The syntax is further simplified since C++17:

```
// Old syntax
auto getTuple() {
    return make_tuple(5, 1.2, 'b');
}
// New syntax since C++17
auto getTuple() {
    return tuple(5, 1.2, 'b');
}
```

Working with Tuples (Cont’d)

- Unpacking the elements is also simplified and no longer requires the `std::tie`.
- The variables `i`, d, and `c` below are declared automatically:

```
int main() {
    auto [i, d, c] = getTuple();
    cout << i; // "5"
}
```

Lambda Functions a.k.a. Anonymous Functions

- Since C++11, we can create **lambda functions, i.e., anonymous function objects**, which provide a compact way to define functions at their point of use, without having to create a named function or function object:

```
auto sum = [](int x, int y) -> int {
    return x + y;
};
cout << sum(2, 3); // "5"
```

Capturing clause

Return value type

Return Values of Lambda Functions

```
auto sum = [](int x, int y) -> int {
    return x + y;
};
```

Function object

Return value

```
auto sum = [](int x, int y) {
    return x + y;
};
```

```
auto sum = [](auto x, auto y) {
    return x + y;
};
```

Common Usage of Lambda Functions

- Specify simple functions that are only referenced once, often by passing the function object as an argument to another function:

```
#include <iostream>
#include <functional>
using namespace std;
void call(int arg, function<void(int)> func)
{ func(arg); }
int main() {
  auto printSquare = [](int x) { cout << x*x; };
  call(2, printSquare); // "4"
}
```

Advantages of Lambda Functions

- Lambdas make code more readable.
- Lambdas improve locality of the code.
- Lambdas allow to store state easily.
- Lambdas allow several overloads in the same place.
- Lambdas get better with each revision of C++.

https://www.cppstories.com/2020/05/lambdasadvantages.html/

Lambdas Make Code More Readable

Without lambdas

```
#include <algorithm>
#include <functional>
#include <vector>
int main() {
    using std::placeholders::_1;
    const std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    const auto val = std::count_if(v.begin(), v.end(),
                          std::bind(std::logical_and<bool>(),
                          std::bind(std::greater<int>(),_1, 2),
                          std::bind(std::less_equal<int>(),_1,6)));

    return val;
}
```

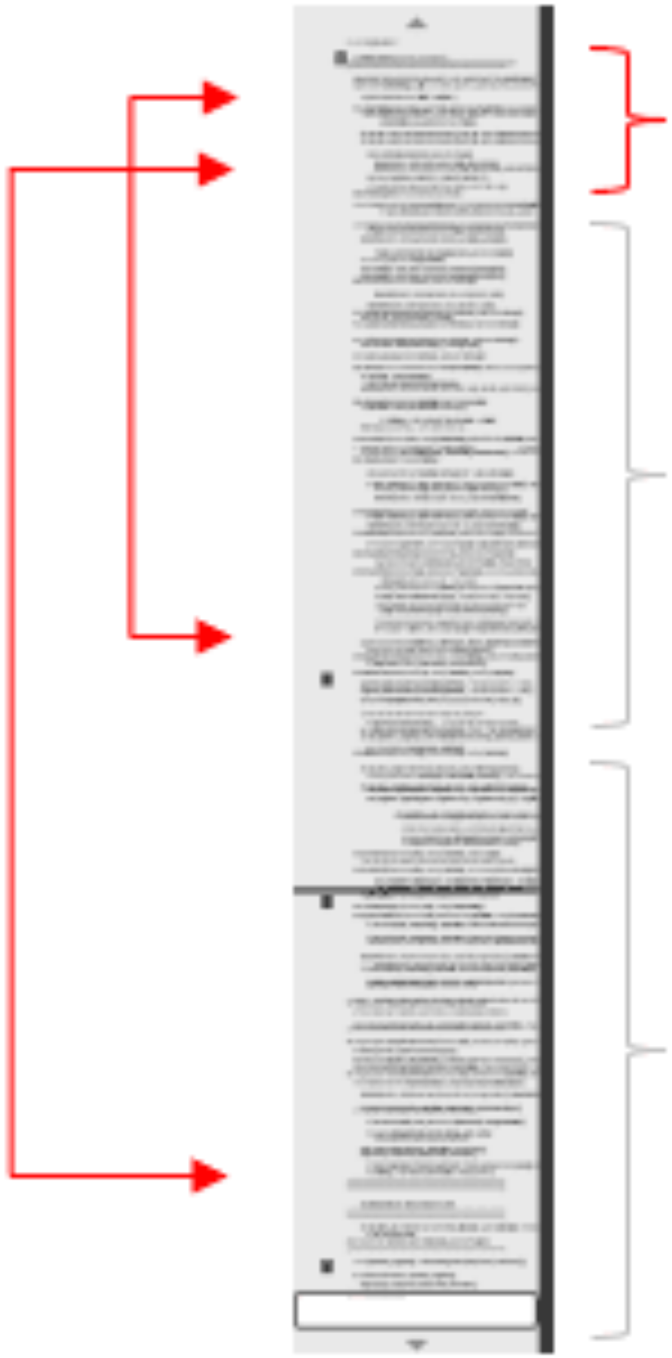
With lambdas

```
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    const auto val = std::count_if(v.begin(), v.end(),
                        [](int v) { return v > 2 && v <= 6;});

    return val;
}
```

https://www.cppstories.com/2020/05/lambdasadvantages.html/

Lambdas Improve Locality of the Code

- Now function invokations are closer to where they are refined.



https://www.cppstories.com/2020/05/lambdasadvantages.html/

Lambdas Allow to Store State Easily

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec { 0, 5, 2, 9, 7, 6, 1, 3, 4, 8 };
    size_t compCounter = 0;
    std::sort(vec.begin(), vec.end(), [&compCounter](int a, int b) {
        ++compCounter;
        return a < b;
    });
    std::cout << "number of comparisons: " << compCounter << '\n';
    for (auto& v : vec)
        std::cout << v << ", ";
}
```

這個 [ ] 裡面可以用來記錄狀態

用來計算做了幾次比較

早…早上好

https://www.cppstories.com/2020/05/lambdasadvantages.html/

Lambdas Allow Several Overloads in the Same Place

- We’ll see example for this later (as it is more advanced).

https://www.cppstories.com/2020/05/lambdasadvantages.html/

Lambdas Get Better with Each Revision of C++!

- Never mind these details… just to let you know that there are many…
  - C++14
    - Generic lambdas - you can pass auto argument, and then the compiler expands this code into a function template.
    - Capture with initialiser - with this feature you can capture not only existing variables from the outer scope, but also create new state variables for lambdas. This also allowed capturing moveable only types.
  - C++17
    - constexpr lambdas - in C++17 your lambdas can work in a constexpr context.
    - Capturing this improvements - since C++17 you can capture *this OBJECT by copy, avoiding dangling when returning the lambda from a member function or store it. (Thanks to Peter Sommerlad for improved wording and checking).
  - C++20
    - Template lambdas - improvements to generic lambdas which offers more control over the input template argument.
    - Lambdas and concepts - Lambdas can also work with constrained auto and Concepts, so they are as flexible as functors as template functions
    - Lambdas in unevaluated contexts - you can now create a map or a set and use a lambda as a predicate.

https://www.cppstories.com/2020/05/lambdasadvantages.html/

Classes

希望可以達成『物件封裝』的概念

class的定義不會分配記憶體？

- A **class** is a template used to create objects.

```
class MyRectangle; // Class declaration
```

✓

class 就是定義出了一個命名空間

```
class MyRectangle { // Class definition
  int x, y;        // member variable / field
  int getArea(); // method / member function
};
```

The getArea of MyRectangle

```
int MyRectangle::getArea() {
  return x * y;
}
```

在 MyRectangle 這個命名空間裡面的 getArea

Inline Methods

inline method 是為了提升程式的執行效率

- Methods of classes can also be inlined:

✓

使用inline之後，編譯器會思考這個函式是不是很短又反覆被呼叫？如果函式很短又反覆被呼叫，編譯器會幫我把函式肚子裡的內容複製到函式出現的地方

  - **Way 1**

```
// Explicit inline methods
inline int MyRectangle::getArea() { return x * y; }
```

  - **Way 2** 新的C++出了這個新的方式

```
class MyRectangle {
   int x, y;
   // Implicit inline methods
   int getArea() { return x * y; }
};
```

Object Creation

- With the class definition, we may create a **class object (/instance)** in the same way that variables are declared.

```
class MyRectangle {…};
int main() {
    MyRectangle r; // object creation
}
```

Access Modifiers

- To reduce the amount of code that must be checked when debugging, one needs to limit the accesses to class members by **access modifiers**:

預設的情況都是 private，因為我們不想要誰都可以存取裡面的資料，避免外漏

  - `private`: Only accessible within the same class. The default setting if no access modifier is explicitly given.

protected：只有子類別可以存取裡面的成員

  - `protected`: Only accessible within the same class and its subclasses.
  - `public`: Accessible by any external code (provided that the class is visible).

- ***For a `class`, members are by default `private`.***
  For a `struct`, members are by default `public`.
  This maintains the *backward compatibility* with C.

這樣設計的原因是為了讓 C語言的程式在C++也大致能跑

Dot Operator . Directly Accesses Object Members

一個自訂的型別

```
class MyRectangle {
public:
```

這樣用「public：」宣告的成員都會是 public

```
  int x, y;
  int getArea() { return x * y; }
```

這是一個 inline method

```
};
...
```

物件.成員名字 = 10;

```
r.x = 10;
r.y = 5;
int z = r.getArea(); // 50 (5*10)
```

. 可以用來存取物件的成員

Arrow Operator ->: Dereference * Then Access .

Dereference：提領運算子

```
MyRectangle r;
MyRectangle *p = &r; // object pointer
r.x = 2;
r.y = 3;
p->getArea(); // 6 (2*3)
(*p).getArea(); // alternative but dumb syntax
```

做了一個 MyRectangle 物件叫做 r

把 x,y 宣告成private 或 protected ，可以確保不會被隨便亂改

做的事情一樣,但上面比較簡潔

*The parenthesis is necessary because . has a higher precedence than *. Why? Since we more often need *(a.b) than (*a).b. This is just a trade-off of the syntax.*

在C++裡這邊要加上（）

Forward Declaration

- Classes and functions must be declared before they can be referenced. If a class definition does not appear before the first reference to that class, a **class prototype** can be specified above the reference instead.

  ```
  class MyClass;
  ```

- This **forward declaration** allows the class to be referenced in any context that does not require the class to be fully defined.

  ```
  class MyClass;
  …
  MyClass* p; // OK
  MyClass f(MyClass&); // OK
  MyClass o; // Error! Definition required
  sizeof(MyClass); // Error! Definition required
  ```

Class Prototype/Declaration vs. Definition

- ***With a class prototype/declaration, we can create pointers to, but not objects of, that class.***
- ***With a class definition, we can create pointers and objects/instances of the class.***
- As a "definition" involves memory space allocation, why is a class definition called "definition"?

  The static member variables, if any, will be allocated memory space in the class definition.

建構子vs解構子

Constructors vs. Destructors: Definition

- A class can contain a **constructor**, which initializes its objects as the objects are created with **`new`**.
  - A class may contain multiple different constructors that take different parameters, so that different objects may be initialized in different ways.
- A class can contain a **destructor**, which performs the cleanup tasks when an object of that class is unneeded and thus destroyed through **`delete`**.
  - The destructor is the *inverse function* of the constructor.
  - A class may only have one destructor without parameters nor return values.

Define a Constructor

```
class MyRectangle {
  public:
  int x, y;
  MyRectangle();
};
// Constructor w/o parameters
MyRectangle::MyRectangle() { x = 10; y = 5; }
```

MyRectangle() 是 MyRectangle 的 constructor

```
// Both calls the constructor
MyRectangle r1; // Class object
MyRectangle* pr2 = new MyRectangle; // Pointer to class object
```

Constructor Overloading

- When a class has 2^+ constructor, the constructors are actually overloaded:

```
class MyRectangle {
public:
  int x, y;
  MyRectangle();
  MyRectangle(int, int);
};
```

這個功能是為了：
有時候想要產生一個設定好的 MyRectangle 的物件，
有時候想要自己定義一個 MyRectangle 物件。

```
// Overloaded constructors
MyRectangle::MyRectangle() { x = 10; y = 5; }
MyRectangle::MyRectangle(int a, int b) { x = a; y = b; }
MyRectangle r; // Calls parameterless constructor
MyRectangle t(2,3); // Calls constructor accepting two integers
```

Constructor Delegation

✓

- C++11 supports **constructor delegation** to reduce code redundancy. That is, a more-sophisticated constructor can call a simpler constructor.

- E.g.: Let a class to have a default way of initialization

```
// Default
MyRectangle::MyRectangle() : MyRectangle(10, 5) {}
```

this Keyword

this 是個指標

- The constructor & other methods of an object are all **instance methods** that can use `this` to access inside of the class body:

```
MyRectangle::MyRectangle(int x, int y) {
  this -> x = x;
  this -> y = y;
}
```

前面的 x 是目前這個物件的 x，後面的 x 是參數。意思是可以用參數的 x 來初始化這個物件裡的 x

這兩個參數x,y 在這裡面的可視範圍比較小

- This help distinguish between *parameters* and *member variables or methods* in the class body.

Field Initialization

- Since C++11, simple fields may also be initialized by using the constructor initializer list.
  - This is the recommended way of assigning default values to fields.
  - The value is automatically assigned before the constructor is run.

```
MyRectangle::MyRectangle(int a, int b) :
   x(a), y(b) {}
```

Field Initialization of References

- **A reference must be set when declared.** Thus, a reference field cannot be set in the constructor body, but must be initialized ***in the class definition*** or ***in the constructor initializer list***.

```
class Foo {
public:
   int x;
   int& ref1 = x;
   int& ref2;
   Foo();
};
Foo::Foo() : ref2(x) {}
```

✓

一定要這樣寫，可以確保 reference 在初始化完成之後一定可以指向一個有效的成員

Default Constructor 預設建構子：什麼都不會做，但讓所有的類別都擁有了建構子

- If no constructors are defined for a class, the compiler will automatically create a **default constructor** w/o parameters when the program compiles.
  - ***With the default constructor, a class can be instantiated even if no constructor has been implemented.***
  - The default constructor will only allocate memory for the object, and does no initialization.
  - Like local variables, fields in C++ are not automatically initialized to their default values. The fields will contain whatever garbage is left in their memory locations until they are explicitly assigned values.
  - If any constructor is provided, the default constructor is disabled to prevent surprises. They can also be added back explicitly (See this page).

Destructor ✓

- The unique **destructor** of a class performs the cleanup tasks as an object is unneeded, which often involves the releasing of dynamic memory allocated by the constructors.
- Like constructors, a default destructor will be automatically provided for each class. ✓

```
class Semaphore {
  bool *sem;
public:
  Semaphore() { sem = new bool; }
  ~Semaphore() { delete sem; }
};
```

Tracing the Lifecycle of Class Objects

Try yourself!

✓

```
#include <cstdio>
struct Tracer {
public:
  Tracer(const char* name) : name{ name } {
    printf("%s constructed.\n", name); }
  ~Tracer() { printf("%s destructed.\n", name); }
private:
  const char* const name;
};
```

我用參數的 name 去初始化 class 裡面的 name

constructor、destructor都沒有回傳值

name 是不能改的東西

是一個常數的字串的常數指標

Tracing the Lifecycle of Class Objects (Cont’d)

Try yourself!

```
#include <cstdio>
struct Tracer { /*…*/ };
static Tracer t1{ "Static variable" };
thread_local Tracer t2{ "Thread-local variable" };
int main() {
  printf("A\n");
  Tracer t3{ "Automatic variable" };
  printf("B\n");
  const auto* t4 = new Tracer{ "Dynamic variable" };
  printf("C\n");
}
```

Output

Try yourself!

```
Static variable constructed.
Thread-local variable constructed.
A
Automatic variable constructed.
B
Dynamic variable constructed.
C
Automatic variable destructed.
Thread-local variable destructed.
Static variable destructed.
```

Special Member Functions

```
class A {
public:
  A() = default; // Explicitly include default constructor
  ~A() = default; // Explicitly include default destructor
  // These will be discussed later…
  // Disable move constructor
  A(A&&) noexcept = delete;
  // Disable move assignment operator
  A& operator=(A&&) noexcept = delete;
  // Disable copy constructor
  A(const A&) = delete;
  // Disable copy assignment operator
  A& operator=(const A&) = delete;
};
```

int a = 10;
這樣初始化的時候是用 copy constructor

✓

4/14 87 end

Object Initialization

- C++ provides a number of different ways to create objects and initialize their fields:

```
class MyClass {
public:
  int i;
  MyClass() = default;
  MyClass(int x) : i(x) {}
};
```

cons reference 和 right value reference 的差別？

物件很簡單（例如一個整數）的時候，直接使用call by value比較快；但當物件複雜的時候，使用call by cons reference 才可以避免產生複製建構子，拖累速度

不需要定義這個constructor，預設的空的constructor會被拿來用

用x 這個參數，直接初始化i 這個成員

額外的初始化動作

```
MyClass a(5); // Direct initialization
MyClass b; // Direct initialization
```

call by value 會發生copy

call by cons reference 這種做法不會

a是個別名，是個reference

```
const MyClass& a = MyClass(); // Value initialization
MyClass&& b = MyClass(); // Value initialization
```

這裡的=並不是設值的動作，

```
MyClass a = MyClass(); // Copy initialization: Copy temporary object to a
MyClass b = a; // Copy initialization: Copy the object a to b
```

Initializing Dynamically Allocated Class Objects

- C++ can also initialize dynamically allocated class objects:

```
class MyClass {
public:
  int i;
  MyClass() = default;
  MyClass(int x) : i(x) {}
};
```

請定義一個constructor或是定義一個物件，他有自訂的constructor還有預設的constructor

new：分配一塊記憶體空間

使用new operator來做initialization

```
…
MyClass* a = new MyClass(); // New initialization: Object pointer
MyClass& b = *new MyClass(); // New initialization: Object reference
…
delete a;
delete &b;
```

Aggregate Initialization

- Aggregate initialization is a compact syntax that initializes every class field using a bracket–enclosed list of initializers, in the same way as can be done with arrays.

```
class MyClass {
public:
  int i;
  MyClass() = default;
  MyClass(int x) : i(x) {}
};
…
// Aggregate initialization: i is initialized to 2
MyClass a = { 2 };
```

Uniform Initialization

- C++11 introduces **uniform initialization** to provide a consistent way to initialize types that work the same for any type.
- This initialization syntax works not just for classes but for any type, including primitives, strings, arrays, and standard library containers such as vector.

```
MyClass a { 3 }; // Uniform initialization: i = 3
MyClass a = { 2 }; // Aggregate initialization: i = 2
```

Example: Uniform Initialization

```
#include <string>
#include <vector>
using namespace std;
int main() {
    int i { 1 };
    string s { "Hello" };
    int a[] { 1, 2 };
    int *p = new int [2] { 1, 2 };
    vector<string> box { "one", "two" };
}
```

在這裡vector不會去管string是怎麼初始化的

const char*

唯讀的C式字串

Initializer-list Constructor

- A class can call a constructor with *a list of dynamic lengths*:

Q.

為什麼編譯器知道auto是int？

因為arg裡面裝的都是int的元素。

為什麼這邊要寫auto？

可以降低程式碼的耦合。去耦合以後這段程式碼就可以重複使用，也更容易維護。

為什麼這邊要用range based for loop不用一般的for loop？

因為一般的for loop 需要知道裡面有幾個元素（要知道要跑幾次），但用range based for loop可以直接印出全部

```
#include <iostream>
using namespace std;

class NewClass {
public:
    NewClass(initializer_list<int>);
};
```

宣告

```
NewClass::NewClass(initializer_list<int> args) {
    for (auto x : args)
        cout << x << " ";
}
```

定義

range based for loop：對於arg裡面的每一個X

x在args裡面，args裡面裝的是什麼，是一堆int 的元素

```
int main() {
    NewClass a { 1, 2, 3 }; // "1 2 3"
}
```

Designated Initializers

static field of class：
屬於這個class，但不是屬於這個class的物件，屬於這個class整體的物件

- We can specify the values of nonstatic fields of a class:

```
class TestClass {
public:
    int a = 1, b = 2; // default values 不初始化他也會是這個值，但稍後可以改這個值
};

int main() {
    TestClass o1 { .a = 3, .b = 4 }; // ok, a = 3, b = 4
    TestClass o2 { .a = 5 }; // ok, a = 5, b = 2
    TestClass o3 { .b = 6 }; // ok, a = 1, b = 6
}
```

Designated Initializers (Cont’d)

- The fields must be initialized in order.

```
int main() {
// error, out of order    要照順序寫，否則會出錯
TestClass o4 { .b = 0, .a = 1 };
// error, designated & non-designated
TestClass o5 { .a = 5, 3 };
}
```

Inheritance

李斯科夫替換法則

- Inheritance allows a class to acquire the members of another class. In the following example, Square inherits from Rectangle:

```
class Rectangle {
public:
    int x, y;
    int getArea() { return x * y; }
};
class Square : public Rectangle {};
```

這邊要表明，是想要存取所有的成員，還是只有proctected的成員

Upcasting

可以向上轉型到基底類別的一個物件

子女類別會包含基底類別有的所有東西

- An object can be **upcast** to its base class, because it contains everything that the base class contains.
- An upcast is performed by assigning the object to either a reference or a pointer of its base class type. In the following example, a Square object is upcast to Rectangle. When using Rectangle's interface, the Square object will be viewed as a Rectangle, so only Rectangle's members can be accessed.

```
Square s;
// reference upcast
Rectangle& r = s;
// pointer upcast
Rectangle* p = &s;
```

```
void setXY(Rectangle& r) {
    r.x = 2; r.y = 3;
}
int main() {
    Square s;
    setXY(s);
}
```

Downcasting

- A `Rectangle` reference or pointer that points to a `Square` object can be **downcast** back to a `Square` object.
- This downcast has to be made explicit since downcasting an actual `Rectangle` to a `Square` is disallowed and may crash the program at runtime:

```
// reference downcast
Square& a = static_cast<Square&>(r);
// pointer downcast
Square* b = static_cast<Square*>(p);
```

Constructor Inheritance

- To make sure the fields in the base class are properly initialized, the parameterless constructor of the base class is automatically called when an object of the derived class is created:

```
#include <iostream>
using namespace std;

class B1 {
public:
    int x;
    B1() : x(5) {}
};

class D1 : public B1 {};
int main() {
    D1 d; // calls parameterless constructors of D1 and B1
    cout << d.x; // "5"
}
```

要建構子女類別的物件的時候，有可能會需要父母類別的一些成員，所以要先把父母類別初始化。

Constructor Delegation

- If there is no default constructor in the base class, the derived class must call an appropriate base class constructor. The call to the base constructor can be made explicitly from the derived constructor, by placing it in the constructor's initializer list. This allows arguments to be passed along to the base constructor.

```
class B2 {
public:
  int x;
  B2(int a) : x(a) {}
};
```

把 X 設為 a

```
class D2 : public B2 {
public:
  // call base constructor
  D2(int i) : B2(i) {}
};
```

Constructor Inheritance

- An alternative solution in this case is to inherit the constructor:

```
class D2 : public B2 {
public:
          B2類別裡面有B2這個成員
  using B2::B2; // inherit all constructors from B2
  int y { 0 };
};
int main() {
  D2 d(3); // call inherited B2 constructor
  cout << d.x; // "3"
}
```

Multiple Inheritance

(diamond shaped inheritance)

- C++ allows a derived class to inherit from more than one base class. This is called multiple inheritance. The base classes are then specified in a comma-separated list:

```
class Person {};
class Employee {};
class Teacher: public Person,
               public Employee {};
```

一個類別同時繼承多個基底類別，行為有衝突的時候該聽誰的？

Hiding Derived Members

- A new method in a derived class `Triangle` can redefine a method in a base class `Rectangle` in order to give it a new, more-specific implementation.

```
class Rectangle {
public:
    int x, y;
    Rectangle(int x, int y) : x(x), y(y) {}
    double getArea() { return x * y; }
};
class Triangle : public Rectangle {
public:
    Triangle(int a, int b) : Rectangle(a,b) {}
    double getArea() { return x * y / 2; }
};
```

Hiding Derived Members (Cont'd)

- If a `Triangle` object is created and the `getArea` method is invoked, the `Triangle`'s version of the method will get called.

```
Triangle t { 2,3 }; // uniform initialization
t.getArea(); // 3 (2*3/2) calls Triangle's version
```

- However, if the `Triangle` is upcast to a `Rectangle`, then `Rectangle`'s version will be called instead.

```
Rectangle& r = t; // upcast
r.getArea(); // 6 (2*3) calls Rectangle's version
```

- That is because ***the redefined method has only hidden the inherited method***. This means that `Triangle`'s implementation is redefined downward in the class hierarchy to any children of `Triangle`, but not upward to the base class.

Overriding Derived Members: `virtual`

- In order to ***redefine a method upward in the class hierarchy***—what is called ***overriding***—the method needs to be declared with the **`virtual`** modifier in the base class. This modifier ***allows the method to be overridden in derived classes***.

```
class Rectangle {
public:
    int x, y;
    virtual int getArea() { return x * y; }
};
```

```
Triangle t { 2,3 };
Rectangle& r = t;
r.getArea(); // 3 (2*3/2) calls Triangle's version
```

The `override` Specifier

- C++11 added the **`override`** specifier, which indicates that ***a method is intended to replace an inherited method***.
- Using this specifier allows the compiler to check that there is a virtual method with that same signature.
- This prevents the possibility of accidentally creating a new virtual method in a derived class.
- It is recommended to always include this specifier when overriding methods.

```
class Triangle : public Rectangle {
public:
  virtual double getArea() override { return x * y / 2; }
};
```

The `final` Specifier

- Another specifier introduced in C++11 is **`final`**. This specifier ***prevents a virtual method from being overridden in derived classes***. It also ***prevents derived classes from using that same method signature***.

```
class Base {
  virtual void foo() final {}
};
class Derived : public Base {
  void foo() {} // error: Base::foo marked as final
};
```

The `final` Specifier (Cont’d)

- The `final` specifier can also be applied to a class to prevent any class from inheriting it.

```
class B final {};
class D : B {}; // error: B marked as final
```

Base Class Scoping

- It is still possible to access a redefined method from a derived class by typing the class name followed by the scope resolution operator.
- This is called base class scoping and can be used to allow access to redefined methods that are any number of levels deep in the class hierarchy.

```
class Triangle : public Rectangle {
public:
  Triangle(int a, int b) { x = a; y = b; }
  int getArea() override { return Rectangle::getArea() / 2; }
};
```

Pure Virtual Functions

- Sometimes a base class knows that all derived classes must implement a certain method, but the base class cannot provide a default implementation for that method.
- The base class can then declare the method as a **pure virtual function** by assigning it the value `0`, to enforce deriving classes to implement this method.

```
class Shape {
public:
  virtual double getArea() = 0; // pure virtual function
};
```

Abstract Classes

- **Abstract class**: a class with some pure **virtual functions.**
  - Abstract classes are incomplete and can’t be instantiated.
  - Abstract classes are used for **upcasting** so that deriving classes can use its interface through pointer/reference type.
- **Interface**: a class with only pure virtual functions.
  - Interfaces define the way of a class to interact with other classes.
  - Interfaces are separately provided by other languages like C# or Java.

Example

```
#include <iostream>

class Shape {
public:
    // pure virtual function
    virtual double getArea() = 0;
};

class Rectangle : public Shape {
public:
    int x = 1, y = 2;
    virtual int getArea() override {
        return x * y;
    }
};
```

s是Shape的reference

```
void printArea(Shape& s) {
    std::cout << s.getArea();
}

int main() {
    Rectangle r;
    printArea(r); // "2"
}
```

4/21(三) end

Access Levels: *Private* Members

```
class MyClass {
public: int myPublic;       // Unrestricted access
protected: int myProtected; // Defining/derived class only
private: int myPrivate;     // Defining class only
    void test() {
        myPublic = 0;       // allowed
        myProtected = 0;    // allowed
        myPrivate = 0;      // allowed
    }
};
```

Access Levels: *Protected* Members

```
class MyChild : public MyClass {
    void test() {
        myPublic = 0; // allowed
        myProtected = 0; // allowed
        myPrivate = 0; // inaccessible
    }
};
```

Access Levels: *Public* Members

```
class OtherClass {
    void test(MyClass& c) {
        c.myPublic = 0; // allowed
        c.myProtected = 0; // inaccessible
        c.myPrivate = 0; // inaccessible
    }
};
```

Access Level Guideline

- **When choosing access levels, use the most restrictive level possible**.
  - The more places a member can be accessed, the more places it can be accessed incorrectly, which makes the code harder to debug. Using restrictive access levels also reduces code coupling, which makes it easier to modify the class without breaking the code for any other programmers using that class.
- **Fields should always be private and only exposed through public or protected getter and setter methods**.
  - This makes it easier to ensure that fields are accessed correctly, as the setter can check that the assigned value is valid for the specific field.
  - By leaving out either the getter or setter method, a field may also be restricted to only write or read access from outside the class.

Example: Getter and Setter

```
class Person {
private:
    int age;
public:
    void setAge(int a) { // Setter
        if (age > 200) age = 200;
        else if (age < 0) age = 0;
        else age = a;
    }

    int getAge() {       // Getter
        return age;
    }
};
```

Friend Classes

- A class can be allowed to access the private and protected members of another class by declaring that class a **friend**.
- The friend is allowed to access all members in the class where the friend is defined, but not the other way around.

```
class MyClass {
    int myPrivate;
    // Give OtherClass access
    friend class OtherClass;
};
```

```
class OtherClass {
    void test(MyClass& c) {
        c.myPrivate = 0; // allowed
    }
};
```

Friend Functions

```
class MyClass;
class OtherClass {
public:
    void test(MyClass& c);
    void test2(MyClass& c);
};
class MyClass {
    int myPrivate;
    friend void OtherClass::test(MyClass&);
};
```

```
void OtherClass::test(MyClass& c) {
    c.myPrivate = 0; // allowed
}
void OtherClass::test2(MyClass& c) {
    c.myPrivate = 0; // not allowed
}
```

Friend Functions (Cont’d)

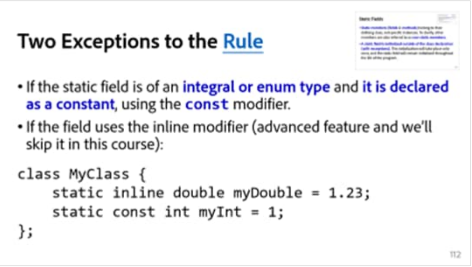
- A global function can also be declared as a friend to a class in order to gain the same level of access:

```
class MyClass {
    int myPrivate;
    // Give myFriend access
    friend void myFriend(MyClass& c);
};
void myFriend(MyClass& c) {
    c.myPrivate = 0; // allowed
}
```

Public, Protected, and Private Inheritance

- When a class is inherited in C++, it is possible to ***change the access level of the inherited members***.
  - **Public inheritance** allows all members to keep their original access level.
  - **Protected inheritance** reduces the access of public members to protected.
  - **Private inheritance** restricts all inherited members to private access.
- ***Private is the default inheritance level, although public inheritance is the one that is nearly always used.***

Static Fields



Two Exceptions to the Rule

- If the static field is of an **integral or enum type** and **it is declared as a constant**, using the `const` modifier.
- If the field uses the inline modifier (advanced feature and we'll skip it in this course):

```
class MyClass {
    static inline double myDouble = 1.23;
    static const int myInt = 1;
};
```

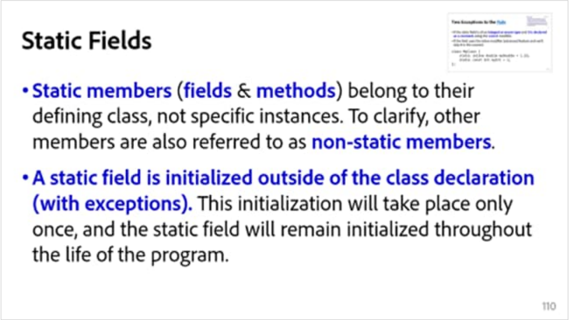
112

- **Static members** (**fields** & **methods**) belong to their defining class, not specific instances. To clarify, other members are also referred to as **non-static members**.
- **A static field is initialized outside of the class declaration (with exceptions).** This initialization will take place only once, and the static field will remain initialized throughout the life of the program.

Example

```
class MyCircle {
public:
    double r; // instance field (one per object)
    static double pi; // static field (only one copy)
};
double MyCircle::pi = 3.14159;

int main() {
    double p = MyCircle::pi;
}
```

Two Exceptions to the Rule



Static Fields

- **Static members** (**fields** & **methods**) belong to their defining class, not specific instances. To clarify, other members are also referred to as **non-static members**.
- **A static field is initialized outside of the class declaration (with exceptions).** This initialization will take place only once, and the static field will remain initialized throughout the life of the program.

- If the static field is of an **integral or enum type** and **it is declared as a constant**, using the **const** modifier.
- If the field uses the inline modifier (advanced feature and we'll skip it in this course):

```
class MyClass {
    static inline double myDouble = 1.23;
    static const int myInt = 1;
};
```

Static Methods

- Methods can also be declared as static.
- A **static method** is not part of any instance, it cannot use instance members as it does not have an implicit this pointer.
- Conventional **instance methods**, in contrast to static methods, can use both static and instance members.

```
class MyCircle {
public:
    double r;
    static inline double pi = 3.14159;
    double getArea() { return pi * r * r; }
    static double newArea(double a) { return pi * a * a; }
};

int main() {
    double a = MyCircle::newArea(1);
}
```

Case Study: a Simple String Class

```
struct SimpleString {
  SimpleString(size_t max_size) : max_size{ max_size }, length{} {
    buffer = new char[max_size];
    buffer[0] = 0;
  }
  ~SimpleString() { delete[] buffer; }
private:
  size_t max_size;
  char* buffer;
  size_t length;
};
```

Try yourself!

Exception Handling: throw

```
#include <stdexcept>
struct SimpleString {

SimpleString(size_t max_size) : max_size{ max_size }, length{} {
  if (max_size == 0) {
    throw std::runtime_error{ "Max size must be at least 1." };
  }
  buffer = new char[max_size];
  buffer[0] = 0;
}

~SimpleString() { delete[] buffer; }
private:
  size_t max_size; // Allocated space
  char* buffer;
  size_t length;   // Actual length of the string; no larger than max_size
};
```

Add the Capabilities of Printing and Appending of Strings

Try yourself!

```
#include <cstdio>
#include <cstring>
#include <stdexcept>
struct SimpleString {
public:
  void print(const char* tag) const {
    printf("%s: %s", tag, buffer);
  }
  bool append_line(const char* x) {
    const auto x_len = strlen(x); // x_len is of type size_t
    if (x_len + length + 2 > max_size) return false;
    // Copy (max_size-length) chars from x to the end of buffer.
    std::strncpy(buffer + length, x, max_size - length);
    length += x_len;
    buffer[length++] = '\n’; // Add a newline char
    buffer[length] = 0;      // End the string
    return true;
  };
}
```

The “Driver” Program

Try yourself!

```
#include <cstdio>
#include <cstring>
#include <exception>
struct SimpleString { /*…*/ }
int main() {
  SimpleString string{ 115 };
  string.append_line("Starbuck, whaddya hear?");
  string.append_line("Nothin' but the rain.");
  string.print("A: ");
  string.append_line("Grab your gun and bring the cat in.");
  string.append_line("Aye-aye sir, coming home.");
  string.print("B: ");
  if(!string.append_line("Galactica!")) {
    printf("String was not big enough to append another message."); {
  }
}
```

Try yourself!

Sample Output

```
A: Starbuck, whaddya hear?
Nothin' but the rain.
B: Starbuck, whaddya hear?
Nothin' but the rain.
Grab your gun and bring the cat in.
Aye-aye sir, coming home.
String was not big enough to append another
message.
```

Construction Order of Members

- A class instance/object may be the static/nonstatic member of another class.
- *Members are constructed before the enclosing object's constructor.*
- *All members are destructed after the object's destructor is invoked.*

Example: Member Construction Order

```
#include <stdexcept>
struct SimpleStringOwner {

  SimpleStringOwner(const char* x) : string{ 10 } {
    if (!string.append_line(x)) {
      throw std::runtime_error{ "Not enough memory!" };
    }
    string.print("Constructed: ");
  }

  ~SimpleStringOwner() {
    string.print("About to destroy: "); v
  }

private:
  SimpleString string;
};
```

Example: Member Construction Order (Cont’d)

```
// Driver
int main() {
SimpleStringOwner x{ "x" };
printf("x is alive\n");
}
```

```
// Output
Constructed: x u
x is alive
About to destroy: x v
```

Copy Semantics

- **Copy semantics** is “the meaning of copy” :
- After x is copied into y, they’re equivalent and independent. That is, x==y is true after a copy (**equivalence**), and a modification to x doesn’t cause a modification of y (**independence**).

Example: Copying Built-in Types

```
#include <cstdio>
int increase_by_one(int x) { return ++x; }
int main() {
  auto original = 1;
  auto result = increase_by_one(original);
  printf("Original: %d; Result: %d", original, result);
}

// Output
Original: 1; Result: 2
```

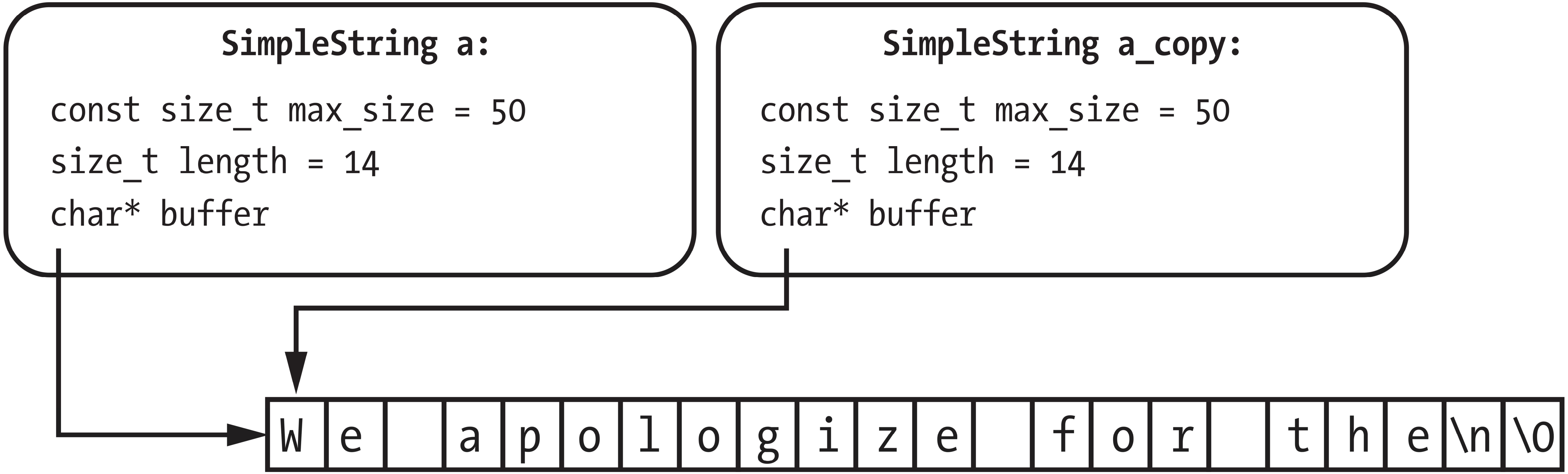
Example: Copying Plain-old-data (POD) Customized Types

```
struct Point { int x, y; };

Point make_transpose(Point p) {
  int tmp = p.x;
  p.x = p.y;
  p.y = tmp;
  return p;
}

// Exercise: Revise the program to swap
// x and y without using temporary variables.
```

Copying (Complex) Class Objects

- **Shallow copy (/member-wise copy)** may not be what we want…

Discussion: What problems will we encounter?



SimpleString a:

```
const size_t max_size = 50
size_t length = 14
char* buffer
```

SimpleString a_copy:

```
const size_t max_size = 50
size_t length = 14
char* buffer
```

W	e		a	p	o	l	o	g	i	z	e		f	o	r		t	h	e	\n	\0

Copying Class Objects, the General Case

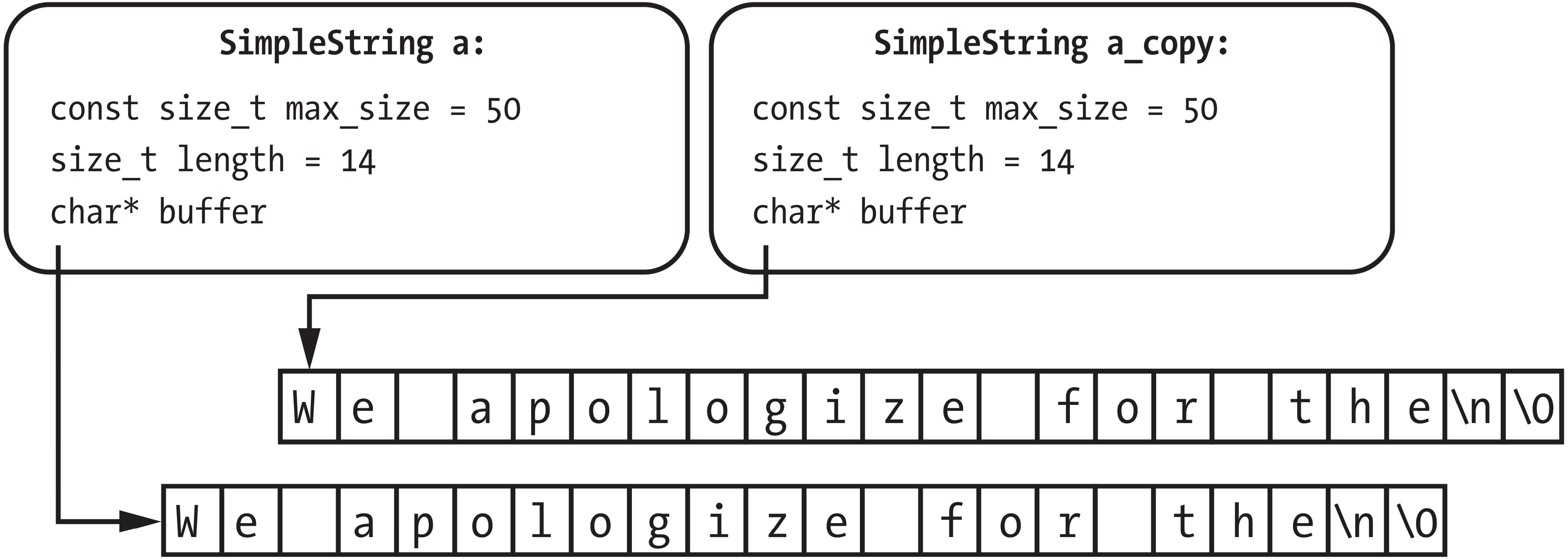
- 2 ways to copy class objects:
  - **Copy construction: Initialize an object using another of the same type through the argument of the constructor.**
  - **Copy assignment construction: Initialize an object by assigning another object of the same type to it.**

Copy Constructor

```
struct SimpleString { /*…*/
SimpleString(const SimpleString& other);
};
int main() {
  SimpleString a;
  SimpleString a_copy{ a }; // Init a_copy with a
}
```

Copying (Complex) Class Objects (Cont'd)

- **Deep copy** copies the data pointed to by the original buffer into a new buffer.



SimpleString a:

```
const size_t max_size = 50
size_t length = 14
char* buffer
```

SimpleString a_copy:

```
const size_t max_size = 50
size_t length = 14
char* buffer
```

W	e		a	p	o	l	o	g	i	z	e		f	o	r		t	h	e	\n	\0

W	e		a	p	o	l	o	g	i	z	e		f	o	r		t	h	e	\n	\0

Example: (Deep) Copy Constructor

```
SimpleString(const SimpleString& other) :
  max_size{ other.max_size },
  buffer{ new char[other.max_size] },
  length{ other.length } {
  std::strncpy(buffer, other.buffer,
    max_size); // Copy the contents of target
}
```

Example: (Deep) Copy Constructor (Cont’d)

```
// Driver
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  SimpleString a_copy{ a };
  a.append_line("inconvenience.");
  a_copy.append_line("incontinence.");
  a.print("a");
  a_copy.print("a_copy");
}
```

```
// Output
a: We apologize for the
inconvenience.
a_copy: We apologize for the
incontinence.
```

Combined Example: Call by Value + Copy Constructor

```
void foo(SimpleString x) {
  x.append_line("This change is lost.");
}
int main() {
  SimpleString a { 20 };
  foo(a); // Invokes copy constructor
  a.print("Still empty"); // Why?
}
```

Rule

- **Do not pass by value to avoid modification. Use a *const reference.***
- Exercise: Why?

Wrong Example w/o Default Copy Assignment Constructor

```
void dont_do_this() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  SimpleString b{ 50 };
  b.append_line("Last message");
  b = a; // Undefined w/o def copy assign constructor
}
```

Copy Assignment Constructor: Skeleton

```
struct SimpleString {
  /*…*/
  SimpleString& operator=(const
    SimpleString& other) { // copy assign
constructor
    if (this == &other) return *this; u
    /*…*/
    return *this; v
  }
}
```

Copy Assignment Constructor: Contents

```
SimpleString& operator=(const SimpleString& other) {
  if (this == &other) return *this;
  const auto new_buffer = new char[other.max_size];
  delete[] buffer;
  buffer = new_buffer;
  length = other.length;
  max_size = other.max_size;
  strcpy_s(buffer, max_size, other.buffer);
  return *this;
}
```

Example

```
// Driver
int main() {
  SimpleString a{ 50 };
  a.append_line("We apologize for the");
  SimpleString b{ 50 };
  b.append_line("Last message");
  a.print("a");
  b.print("b");
  b = a; // Invoke the copy assignment constructor
  a.print("a");
  b.print("b");
}
```

```
// Output
a: We apologize for the
b: Last message
a: We apologize for the
b: We apologize for the
```

Exercise: Default Copy Assignment Constructor

- Default copy assignment constructors use shallow copy.
- **Phase 1.**
  - Verify this by writing a class called A with a pointer field p pointing to dynamically allocated memory.
  - Use default copy assignment operator to verify that the default copy assignment constructors use *shallow copy*.
- **Phase 2.**
  - Implement a deep copy constructor for A and verify its correctness.
- Compress and upload your code (no executable please) to Google Drive and send me the link through Zuvio. (Turn on the download privilege please!)

Thank You Very Much!

Q&A?