Lecture Participation Poll #21

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 21: C++ Objects

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

- HW5 due date moved to Wed Dec 1
- Office hours will shift a little next week

Malloc vs New

	`malloc()`	`new`
What is it?	a function	an operator or keyword
How often used (in C)?	often	never
How often used (in C++)?	rarely	often
Allocated memory for	anything	arrays, structs, objects, primitives
Returns	a `void*` (*should be cast*)	appropriate pointer type (*doesn't need a cast*)
When out of memory	returns `NULL`	throws an exception
Deallocating	`free()`	`delete` or `delete[]`

Dynamically Allocated Arrays

- To dynamically allocate an array:

```
type* name = new type[size];
```

  - calls default (zero-argument) constructor for each element
  - convenient if there's a good default for initialization

- To dynamically deallocate an array:
  - Use `delete[] name;`
  - It is an *incorrect* to use "delete name;" on an array
    - The compiler probably won't catch this, though (**!**) because it can't always tell if name* was allocated with new type[size]; or new type;
    - Especially inside a function where a pointer parameter could point to a single item or an array and there's no way to tell which!
    - Result of wrong delete is undefined behavior

Arrays Example (Primitives)

arrays.cpp

```
#include "Point.h"

int main() {
  int stack_int;
  int* heap_int = new int;
  int* heap_int_init = new int(12);

  int stack_arr[3];
  int* heap_arr = new int[3];

  int* heap_arr_init_val = new int[3]();
  int* heap_arr_init_lst = new int[3]{4, 5};  // C++11

  ...

  delete heap_int;             //
  delete heap_int_init;        //
  delete heap_arr;             //
  delete[] heap_arr_init_val;  //

  return EXIT_SUCCESS;
}
```

Arrays Example (Objects)

arrays.cpp

```
#include "Point.h"

int main() {
  ...

  Point stack_pt(1, 2);
  Point* heap_pt = new Point(1, 2);

  Point* heap_pt_arr_err = new Point[2];

  Point* heap_pt_arr_init_lst = new Point[2]{{1, 2},
{3, 4}};
                                                       /
/ C++11
  ...

  delete heap_pt;
  delete[] heap_pt_arr_init_lst;

  return EXIT_SUCCESS;
}
```

Pointers in C++

- Work the same as in C, hooray!
- A **pointer** is a variable containing an address
  - Modifying the pointer *doesn't* modify what it points to, but you can access/modify what it points to by *dereferencing*

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int* z = &x;

  *z += 1;  // sets x to 6
   x += 1;  // sets x (and *z) to 7

   z = &y;  // sets z to the address of y
  *z += 1;  // sets y (and *z) to 11

  return EXIT_SUCCESS;
}
```

References in C++

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
  - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // sets z (and x) to the value of y
  z += 1;  // sets z (and x) to 11

  return EXIT_SUCCESS;
}
```

Pass by Reference

C++ allows you to use real **pass-by-reference**

- Client passes in an argument with normal syntax
  - Function uses reference parameters with normal syntax
  - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

- In C all function arguments are copies
- pointer arguments pass a copy of the address value, original values will be unaffected by changes to parameter

- A stylistic choice, not mandated by the C++ language
- Google C++ style guide suggests:
  - Input parameters:
    - Either use values (for primitive types like int or small structs/objects)
    - Or use const references (for complex struct/object instances)
  - Output parameters:
    - Use unchangeable pointers referencing changeable data
  - Ordering:
    - List input parameters first, then output parameters last

Structs in C vs Classes in C++

- In C, a struct can only contain data fields
  - No methods and all fields are always accessible
- In C++, struct and class are (nearly) the same!
  - Both can have methods and member visibility (public/private/protected)
  - <u>Minor difference</u>: members are **default *public*** in a struct and **default *private*** in a class
  - structs need to allocate heap memory so object will persist
- Common style convention:
  - Use struct for simple bundles of data
  - Use class for abstractions with data + functions

Classes in C++

- Unlike C structs
  - Class definition is part of interface and should go in .h file
    - Private members still must be included in definition (**!**)
  - Typically put member function definitions into companion .cpp file with implementation details
    - Common exception: setter and getter methods
  - These files can also include non-member functions that use the class
- Like java
  - Fields & methods, static vs instance, constructors
  - method overloading (functions, operators and constructors)
- Not quite like Java
  - access-modifier (eg private) syntax
  - declaration separate from implementation (like C)
  - funny constructor syntax, default parameters (eg, …=0)
- Not at all like Java
  - you can name files anything you want
    - Typically a combination of Name.cpp and Name.h for class Name
  - destructors and copy constructors
  - virtual vs non-virtual

MyClass.h

```
namespace mynamespace {
   class MyClass {
      private:
         type fieldOne;
         type fieldTwo;

      public:
         MyClass();
         MyClass(type, type);

      public:
         type functionOne() {
            // function definition
         }
         type functionTwo() {
            // function definition
         }
   };
}
```

Defining Classes in C++

- Class Definition (in a .h file)

Name.h

```
class Name {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
};  // close class Name
```

- Class Member Definition (in a .cpp file)

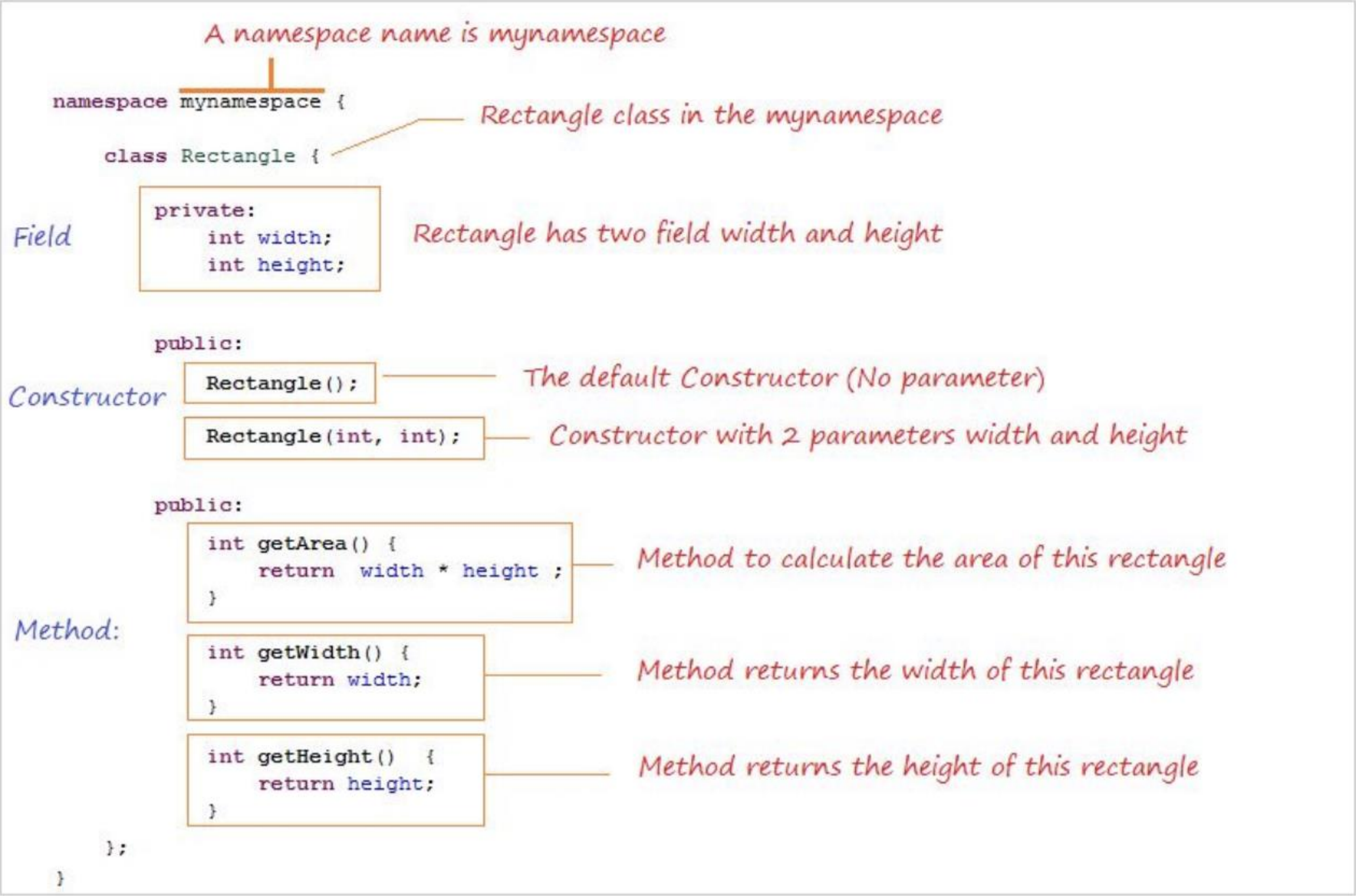
Name.cpp

```
returnType ClassName::MethodName(type1 param1, …, typeN paramN) {
  // body statements
}
```

- Members can be functions (methods) or data (variables)
- (1) *define* within the class definition OR (2) *declare* within the class definition and then *define* elsewhere

Anatomy of C++ Class

Rectangle.h



```
namespace mynamespace {

    class Rectangle {

        private:
            int width;
            int height;


        public:
            Rectangle();
            Rectangle(int, int);


        public:
            int getArea() {
                return  width * height ;
            }
            int getWidth() {
                return width;
            }
            int getHeight()  {
                return height;
            }
    };
}
```

A namespace name is mynamespace

Rectangle class in the mynamespace

Field

Rectangle has two field width and height

Constructor

The default Constructor (No parameter)

Constructor with 2 parameters width and height

Method:

Method to calculate the area of this rectangle

Method returns the width of this rectangle

Method returns the height of this rectangle

Access Control

- Access modifiers for members:
  - public: accessible to *all* parts of the program
  - private: accessible to the member functions of the class
    - Private to *class*, not object instances
  - protected: accessible to member functions of the class and any *derived* classes (subclasses – more to come, later)
- Reminders:
  - Access modifiers apply to *all* members that follow until another access modifier is reached
  - If no access modifier is specified, struct members default to public and class members default to private

Class Definition (Member declaration)

Point.h

```
#ifndef POINT_H_
#define POINT_H_

class Point {
 public:
  Point(const int x, const int y); // constructor
  int get_x() const { return x_; } // inline member function
  int get_y() const { return y_; } // inline member function
  double Distance(const Point& p) const; // member function
  void SetLocation(const int x, const int y); // member function

 private:
  int x_;   // data member
  int y_;   // data member
};  // class Point

#endif  // POINT_H_
```

Class Member Definition

Point.cpp

```
#include <cmath>
#include "Point.h"

Point::Point(const int x, const int y) {
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

double Point::Distance(const Point& p) const {
  // We can access p’s x_ and y_ variables either through the
  // get_x(), get_y() accessor functions or the x_, y_ private
  // member variables directly, since we’re in a member
  // function of the same class.
  double distance = (x_ - p.get_x()) * (x_ - p.get_x());
  distance += (y_ - p.y_) * (y_ - p.y_);
  return sqrt(distance);
}

void Point::SetLocation(const int x, const int y) {
  x_ = x;
  y_ = y;
}
```

Class Usage

usePoint.cpp

```
#include <iostream>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
  Point p1(1, 2);  // allocate a new Point on the Stack
  Point p2(4, 6);  // allocate a new Point on the Stack

  cout << "p1 is: (" << p1.get_x() << ", ";
  cout << p1.get_y() << ")" << endl;

  cout << "p2 is: (" << p2.get_x() << ", ";
  cout << p2.get_y() << ")" << endl;

  cout << "dist : " << p1.Distance(p2) << endl;
  return 0;
}
```

To allocate on the heap use the “new” keyword

```
Point* p1 = new Point(1, 2);
```

Constructors in C++

- A constructor initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
    - Which one is invoked depends on *how* the object is instantiated
- Written with the class name as the method name:

```
Point(const int x, const int y);
```

  - C++ will automatically create a synthesized default constructor if you have *no* user-defined constructors
    - Takes no arguments and calls the default constructor on all non-"plain old data" (non-POD) member variables
    - Synthesized default constructor will fail if you have non-initialized const or reference data members
- 4 different types of constructors
  - default constructor – takes zero arguments. If you don't define any constructors the compiler will generate one of these for you (just like Java)
  - copy constructor – takes a single parameter which is a *const reference*(`const T&`) to another object of the same type, and initializes the fields of the new object as a *copy* of the fields in the referenced object
  - user-defined constructors – initialize fields and take whatever arguments you specify
  - conversion constructors – implicit, take a single argument. If you want a single argument constructor that is not implicit must use the keyword "explicit" like: `explicit String(const char* raw);`

Synthesized Default Constructor

SimplePoint.h

```
class SimplePoint {
 public:
  // no constructors declared!
  int get_x() const { return x_; }      // inline member function
  int get_y() const { return y_; }      // inline member function
  double Distance(const SimplePoint& p) const;
  void SetLocation(int x, int y);

 private:
  int x_;   // data member
  int y_;   // data member
};  // class SimplePoint
```

SimplePoint.cpp

```
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;  // invokes synthesized default constructor
  return EXIT_SUCCESS;
}
```

Synthesized Default Constructor

- If you define *any* constructors, C++ assumes you have defined all the ones you intend to be available and will *not* add any others

```
#include "SimplePoint.h"

// defining a constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
  x_ = x;
  y_ = y;
}

void foo() {
  SimplePoint x;         // compiler error:  if you define any
                         // ctors, C++ will NOT synthesize a
                         // default constructor for you.

  SimplePoint y(1, 2);   // works:  invokes the 2-int-arguments
                         // constructor
}
```

Overloading Constructors

```
#include "SimplePoint.h"

// default constructor
SimplePoint::SimplePoint() {
  x_ = 0;
  y_ = 0;
}

// constructor with two arguments
SimplePoint::SimplePoint(const int x, const int y) {
  x_ = x;
  y_ = y;
}

void foo() {
  SimplePoint x;          // invokes the default constructor
  SimplePoint y(1, 2);    // invokes the 2-int-arguments ctor
  SimplePoint a[3];       // invokes the default ctor 3 times
}
```

Copy Constructors

- C++ has the notion of a copy constructor
  - Used to create a new object as a copy of an existing object
  - Initializer lists can also be used in copy constructors
  - initializes a new bag of bits (new variable or parameter)
  - assignment (=) replaces an existing value with a new one
    - may need to clean up old state (free heap data?)

```
Point::Point(const int x, const int y) : x_(x), y_(y) { }

// copy constructor
Point::Point(const Point& copyme) {
  x_ = copyme.x_;
  y_ = copyme.y_;
}

void foo() {
  Point x(1, 2);   // invokes the 2-int-arguments constructor

  Point y(x);      // invokes the copy constructor
  Point z = y;     // also invokes the copy constructor
}
```

Synthesized Copy Constructor

- If you don't define your own copy constructor, C++ will synthesize one for you
  - It will do a *shallow* copy of all of the fields (*i.e.* member variables) of your class
  - Sometimes the right thing; sometimes the wrong thing

```
#include "SimplePoint.h"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x;
  SimplePoint y(x);  // invokes synthesized copy constructor
  ...
  return EXIT_SUCCESS;
}
```

When Do Copies Happen?

- The copy constructor is invoked if:
  - You *initialize* an object from another object of the same type:

```
Point x;        // default ctor
Point y(x);     // copy ctor
Point z = y;    // copy ctor
```

  - You pass a non-reference object as a value parameter to a function:

```
void foo(Point x) { ... }

Point y;        // default ctor
foo(y);         // copy ctor
```

  - You return a non-reference object value from a function:

```
Point foo() {
  Point y;      // default ctor
  return y;     // copy ctor
}
```

Initialization Lists

- C++ lets you *optionally* declare an initialization list as part of a constructor definition
  - Initializes fields according to parameters in the list
  - The following two are (nearly) identical:

```
Point::Point(const int x, const int y) {
  x_ = x;
  y_ = y;
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

```
// constructor with an initialization list
Point::Point(const int x, const int y) : x_(x), y_(y) {
  std::cout << "Point constructed: (" << x_ << ",";
  std::cout << y_<< ")" << std::endl;
}
```

Initialization vs Construction

- Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering
  - Data members that don't appear in the initialization list are *default initialized/constructed* before body is executed
- Initialization preferred to assignment to avoid extra steps
  - Never mix the two styles

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(const int x, const int y, const int z) : y_(y), x_(x) {
    z_ = z;
  }

 private:
  int x_, y_, z_;   // data members
};  // class Point3D
```

First, initialization list is applied.

Next, constructor body is executed.

Destructors

- C++ has the notion of a destructor
  - Like “free” in c. In fact, invokes free under the hood to clean up when freeing memory
  - Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)
    - Do not need to call destructors explicitly
  - Place to put your cleanup code – free any dynamic storage or other resources owned by the object
  - Standard C++ idiom for managing dynamic resources
    - Slogan: “*Resource Acquisition Is Initialization*” (RAII)

```
Point::~Point() {    // destructor
  // do any cleanup needed when a Point object goes away
  // (nothing to do here since we have no dynamic resources)
}
```

Nonmember Functions

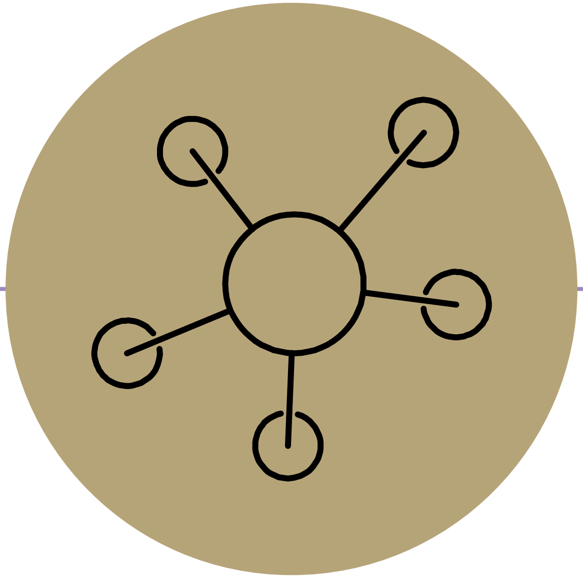
- "Nonmember functions" are just normal functions that happen to use some class
  - Called like a regular function instead of as a member of a class object instance
  - These do *not* have access to the class' private members
- Useful nonmember functions often included as part of interface to a class
  - Declaration goes in header file, but *outside* of class definition
- A class can give a nonmember function (or class) access to its non-public members by declaring it as a **friend** within its definition
  - Not a class member, but has access privileges as if it were
  - friend functions are usually unnecessary if your class includes appropriate "getter" public functions

```
class Complex {
  ...
  friend std::istream& operator>>(std::istream& in, Complex& a);
  ...
};  // class Complex
```

Complex.h

Complex.cpp

```
std::istream& operator>>(std::istream& in, Complex& a)
{
  ...
}
```



Questions

RAII

- "Resource Acquisition is Initialization"
- Design pattern at the core of C++
- When you create an object, acquire resources
  - Create = constructor
  - Acquire = allocate (e.g. memory, files)
- When the object is destroyed, release resources
  - Destroy = destructor
  - Release = deallocate
- When used correctly, makes code safer and easier to read

```
char* return_msg_c() {
  int size = strlen("hello") + 1;
  char* str = malloc(size);
  strncpy(str, "hello", size);
  return str;
}
```

```
std::string return_msg_cpp() {
  std::string str("hello");
  return str;
}
```

```
using namespace std;
char* s1 = return_msg_c();
cout << s1 << endl;
string s2 = return_msg_cpp();
cout << s2 << endl;
```

Compiler Optimization

- The compiler sometimes uses a "return by value optimization" or "move semantics" to eliminate unnecessary copies
  - Sometimes you might not see a constructor get invoked when you might expect it

```
Point foo() {
  Point y;            // default ctor
  return y;           // copy ctor? optimized?
}

Point x(1, 2);      // two-ints-argument ctor
Point y = x;        // copy ctor
Point z = foo();    // copy ctor? optimized?
```

Namespaces

- Each namespace is a separate scope
  - Useful for avoiding symbol collisions!
- Namespace definition:

  -
    ```
    namespace name {
      // declarations go here
    }
    ```
  - Doesn't end with a semi-colon and doesn't add to the indentation of its contents
  - Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace* (**!**)
    - This means that components (*e.g.* classes, functions) of a namespace can be defined in multiple source files
- Namespaces vs classes
  - They seems somewhat similar, but classes are *not* namespaces:
  - There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)
  - To access a member of a namespace, you must use the fully qualified name (*i.e.* nsp_name::member)
    - Unless you are using that namespace
    - You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition

Const

- C++ introduces the “const” keyword which declares a value that cannot change
- const int CURRENT_YEAR = 2020;