C/C++ Programming
Southern University of Science and Technology
Mengxuan Wu
12212006

Final Review

Mengxuan Wu

1 Data Types

In C++, we have two build-in data types: fundamental data types and compound data types. Fundamental data types include: **int, float, double, char, bool, void**. Compound data types include: **array, pointer, reference, string, structure, union, enumeration, class, function, etc.** To be noticed that **void** is a special data type.

1.1 Naming Limit

- A variable name can be of any length.
- A variable name must begin with a letter or an underscore character.
- A variable name can only contain letters, numbers, and the underscore character.
- Variable names are case-sensitive.
- A name that begins with two underscores or an underscore followed by an uppercase letter is reserved for the compiler.
- Reserved words (such as C++ keywords, like int) may not be used as variable names.

1.2 Sizeof Operator

The `sizeof` operator returns the size of a variable or data type. We can use the type name or variable name to get the size of a type or variable. For example:

```
int a;
sizeof(int); // returns 4
sizeof(a); // returns 4
```

1.3 Data Initialization

There are several ways to initialize a variable in C++.

- `int a = 0;`
- `int a(0);`

- `int a{0};`
- `int a = {};`
- `int a = {0};`

To be noticed that if we initialize a variable without a value in main function, it will be initialized to a random value. But if we initialize a static variable without a value, it will be initialized to 0 (or default value).

1.4 Integer Types

Type	Size	Range
char	1 byte	-2^7 to 2^7-1 or 0 to 2^8-1
short	2 bytes	-2^{15} to $2^{15}-1$ or 0 to $2^{16}-1$
int	4 bytes	-2^{31} to $2^{31}-1$ or 0 to $2^{32}-1$
long	4 bytes	-2^{31} to $2^{31}-1$ or 0 to $2^{32}-1$
long long	8 bytes	-2^{63} to $2^{63}-1$ or 0 to $2^{64}-1$

Table 1: Integer Types

However, data width largely depends on the compiler and the computer architecture. The true standard is that: **int** is at least 16 bits, **long** is at least 32 bits, and **long long** is at least 64 bits.

The compiler will provide the **limits.h** header file, which defines macros that allow you to use these values and other details about the binary representation of integer values in your programs. For example **INT_MIN** and **INT_MAX**.

1.4.1 Prefix for Different Bases

We might want to use different bases to represent an integer. This can be specified by adding a prefix to the integer literal. If we want to use octal, we can add a **0** prefix (numbers like 09 will trigger an error). If we want to use hexadecimal, we can add a **0x** or **0X** prefix. If we want to use binary, we can add a **0b** or **0B** prefix.

1.4.2 Suffix for Different Types

We can also use suffix to specify the type of an integer literal. If we want to use **unsigned**, we can add a **u** or **U** suffix. If we want to use **long**, we can add a **l** or **L** suffix. If we want to use **long long**, we can add a **ll** or **LL** suffix.

1.5 Boolean Type

The **bool** type is used to represent boolean values. Any non-zero value is considered true, even negative numbers.

1.6 Floating-Point Types

Type	Size	Range
float	4 bytes	about 6 digits
double	8 bytes	about 15 digits
long double	8 bytes	about 15 digits

Table 2: Floating-Point Types

The compiler will provide the **float.h** header file, which defines macros that allow you to use these values and other details about the binary representation of floating-point values in your programs.

We can use **E notation** to represent a floating-point number. For example, **1.23e-6** represents 1.23×10^{-6}.

To be noticed, `double a = .2` is a valid statement, and it will initialize `a` to 0.2.

1.7 Const Qualifier

The **const** qualifier can be used to make a variable immutable. You can only use **const** qualifier when you declare a variable. For example:

```
const int a = 0; // correct

const int b;     // will trigger an error
b = 0;
```

1.8 Arithmetic Operators

In C++, the arithmetic operators have left-to-right associativity, which means `12 / 3 * 4` is equivalent to `(12 / 3) * 4`, instead of `12 / (3 * 4)`.

When division is performed, if there exists one operand that is floating-point type, the result will be a floating-point type.

1.9 Type Conversion

The compiler will perform type conversion by the following rules:

1. If either operand is a floating type, the two operands will be converted to the higher precision type.
2. Otherwise, if both are signed or both are unsigned, the operand with the smaller type will be converted to the larger type.
3. Otherwise, if the unsigned type is larger than the signed type, the signed type will be converted to the unsigned type (if it is a negative number, this will cause an overflow).
4. Otherwise, if the signed type can represent all values of the unsigned type, the unsigned type will be converted to the signed type.

5. Otherwise, the signed type will be converted to the unsigned type.

We can force a type cast by using the following syntax:

```
int a = 0;
double b = (double) a;
double c = double (a);
auto d = double (a);
```

To be noticed, when using an expression to initialize a variable, the order is to first go through the expression and then perform type conversion. For example:

```
int a = 3.5 + 1.5; // a = 5
double b = 3 / 2;   // b = 1.0
```

2 Compound Data Types

2.1 Array

2.1.1 Array Size

The size of an array must be a constant expression (could be 0). If you use `sizeof` operator on the array name, it will return the size of the array in bytes. For example, an `int` array with 10 elements will return 40.

2.1.2 Array Initialization

We use curly braces to initialize an array. This will initialize the array from the first element to the last element. If some elements are not given a value, they will be initialized to 0. However, this is only valid when declaration, not assignment. For example:

```
int a[10] = {1, 2, 3}; // a = {1, 2, 3, 0, 0, 0, 0, 0, 0, 0}
int b[3];
b = {1, 2, 3};         // will trigger an error
```

You can drop the = or omit the array size, if you initialize the array this way. For example:

```
int a[] = {1, 2, 3};  // a = {1, 2, 3}
int b[3]  {1, 2, 3};  // b = {1, 2, 3}
int c[]   {1, 2, 3};  // c = {1, 2, 3}
```

You cannot use an array to initialize another array. For example:

```
int a[10] = {1, 2, 3};
int b[10] = a; // will trigger an error
int c[10];
c = a;         // will trigger an error
```

You cannot use auto-conversion, if you initialize an array this way. For example:

```
char a[1] = {353}; // will trigger an error
char b = 353;      // correct, b = 'a'
char c = {353};    // will trigger an error
char d(353);       // correct, d = 'a'
```

2.2 String

A C-style string is a char array that is terminated by a null character (`'\0'`, whose ascii code is 0). You can use a string to initialize a char array, and the compiler will automatically add a null character at the end (the null character takes up one element and will be counted by `sizeof` operator). When array is not big enough to hold the string, the compiler will trigger an error.

You cannot initialize a single char with a string, even if the string only contains one character or no character. For example:

```
char a = "a"; // will trigger an error
char b = "";  // will trigger an error
char c = 'a'; // correct
```

C++ provides a **string** class to handle strings. Its value can be accessed like an array, and can use `+` or `+=` operator to concatenate strings. If you want to find the length of a string, you can use `.size()` method (this excludes the null character).

2.3 Structure

A structure is a compound data type that groups related data together. You can use curly braces to initialize a structure (partial initialization is allowed). You can even use nested curly braces to initialize an array of structures. For example:

```
struct Node {
    int data;
    int name;
};
Node nodes[2] = {{1, 2}, {3, 4}};
```

A structure can have member functions. For example:

```
struct Node {
    int data;
    int name;
    void print() {
        cout << data << " " << name << endl;
    }
};
```

2.4 Union

A union is a compound data type that allows you to store different data types in the same memory location. But only one member can contain a value at any given time.

2.5 Enumeration

The `enum` type in C++ is used to define a set of named constants. If not specified, the first constant will be assigned a value of 0, and the value of each successive constant will be increased by 1. If any constant is assigned a value, the value of each successive constant will be increased by 1 from the previous constant. To be noticed, these constants can have the same value. For example:

```
enum Color {
    RED = 1,
    GREEN,
    BLUE = 5,
    YELLOW = 5
};
```

This will lead to the following values: `RED = 1, GREEN = 2, BLUE = 5, YELLOW = 5`.

Due to this potential discontinuity, you cannot perform arithmetic operations on `enum` type. But you can cast it to an integer type, and then perform arithmetic operations.

Enum type variable are actually stored as integers. When you compare two `enum` type variables, you are actually comparing their values. And you can initialize an `enum` type variable with an integer value, even if there is no corresponding constant. For example:

```
enum Color {
    ... // same as above
};
Color a = 1;          // wrong
Color b = (Color) 1;  // correct
Color c = Color(1);   // correct
Color d = Color(200); // correct even if there is no corresponding constant
```

3 Pointer

To get the address of a variable, we can use the `&` operator. Conversely, to get the value of a variable from its address, we can use the `*` operator.

3.1 Declaration and Initialization

For example, to declare a pointer-to-int variable `p` and initialize it to point to variable `a`, we can use the following syntax:

```
int a = 0;
int * p = &a;
```

Two spaces are optional in the above syntax: The space between `int` and `*` and the space between `*` and `p`.

The content of a pointer variable is the address of another variable. To assign a value to a pointer variable is dangerous, because it means you let it points to a memory location that you do not know what it is. For example, the following code is correct, but it is dangerous:

```
int *p = (int *) 0xB8000000;
```

3.2 Initialization with `new`

We can use `new` operator to dynamically allocate memory, and use `delete` operator to free the memory.

```
int *p = new int;
int *q = new int[10];
delete p;
delete[] q;
```

There are several things to be noticed:

- You cannot delete a memory location that is not allocated by `new`.
- You cannot delete a memory location that is already deleted.
- It is safe to delete a `nullptr`.
- If you try to delete a dynamically allocated array, make sure the pointer points to the first element of the array.

3.3 Dynamic Array

We can use `new` operator to dynamically allocate an array.
To iterate through a dynamic array, we simply increment the pointer. For example:

```
int *p = new int[10];
for (int i = 0; i < 10; i++) {
    cout << p[i] << endl; // correct
}
for (int i = 0; i < 10; i++) {
    cout << *p << endl;   // also correct
    p++;
}
p -= 10; // reset p
delete [] p;
```

Also, this works on normal arrays. For example:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = a;                                    // or int *p = &a[0];
for (int i = 0; i < 10; i++) {
    cout << *p << endl; // correct
    p++;
}
```

Especially, a dynamic array is useful for multiple strings. You can declare a dynamic array of `char *` to store multiple strings. For example:

```
const char * names[] = {"Alice", "Bob", "Cindy"};
```

The array initialized in this way will not be forced to be aligned at the end.

3.4 Dynamic Structure

We can use `new` operator to dynamically allocate a structure. If we want to access the members of the structure, we can use the `->` operator.

3.5 Pointer and Const

3.5.1 Const Pointer

A const pointer is a pointer whose value (the address it holds) cannot be changed after initialization. But you can change the value of the variable it points to. For example:

```
int a = 0;
int b = 1;
int * const p = &a;
p* = 1; // correct
p = &b; // will trigger an error
```

3.5.2 Pointer to Const

A pointer to const is a pointer that points to a const variable. You cannot change the value of the variable it points to, but you can change the address it holds. For example:

```
int a = 0;
int b = 1;
const int * p = &a; // or int const * p = &a;
p* = 1; // will trigger an error
p = &b; // correct
```

To be noticed, a pointer that points to a const variable must be a pointer to const. For example:

```
const int a = 0;
int * p1 = &a; // will trigger an error
const int * p2 = &a; // correct
```

This is part of const-correctness, which means that even if the pointer may not try to change the value of the variable it points to, it should still be a pointer to const.

4 Storage

4.1 Automatic Storage

Ordinary variables are stored in stack. They will be destroyed when the function terminates.

4.2 Static Storage

You can create a static variable by using the `static` keyword or defining it outside any function. They will exist throughout the lifetime of the program.

4.3 Dynamic Storage

The variables created by `new` operator are stored in heap. They are not tied to any function, and will exist until you delete them.

5 Increment and Decrement

6 Function

6.1 Function Prototype

A function prototype is a declaration of a function that specifies the function's name and type signature (arity, data types of parameters, and return type), but omits the function body. It is necessary to declare a function prototype before you use it, but you don't have to define it.

When declaring a function prototype, you can omit the parameter names. For example:

```
int add(int, int);
```

6.2 Passing Array to Function

When passing an array to a function, you can use the following syntax:

```
void print(int a[]) {
...
}
void print(int *a) {
...
}
```

To be noticed that `a[]` here works the same as `*a`, which means you can perform pointer arithmetic on it. However, this is not possible for a normal array variable. For the very same reason, if you apply `sizeof` operator on `a[]` in the function, it will return the size of a pointer, not the size of the array.

6.2.1 Two Dimensional Array

When passing a two dimensional array to a function, you can use the following syntax:

```
void print(int a[][10]) {
...
}
void print(int (*a)[10]) {
...
}
```

The second size of the array must be specified.

If you apply `sizeof` operator on `a` in the function, it will return the size of a pointer, not the size of the array. But if you apply `sizeof` operator on `a[0]` in the function, it will return the size of the first row of the array.

6.3 Pointer to Function

A pointer to function is a pointer that points to a function. To declare a pointer to function, you can use the following syntax:

```
int add (int, int);
int (*p) (int, int) = add;

p(1, 2);
(*p)(1, 2); // both are correct
```

Notice that the parameter type and number and the return type of the function must match the type of the pointer to function. And this is where the void type is useful (when the return type of the function is void).

6.4 Inline Function

An inline function is a function that is expanded in line when it is called. When the inline function is called, the compiler will replace the function call with the function code.

This will reduce the overhead of a function call, because the program does not have to jump to another location to execute the function code. But it will increase the memory usage, because the function code will be copied to every place where the function is called.

6.5 Passing by Reference

When passing a variable to a function, the default behavior is to pass by value. This means that the function will create a copy of the variable, and any changes to the variable inside the function will not affect the original variable.

However, to actually modify the original variable, we can pass by pointer or pass by reference.

A reference must be initialized by a variable, and cannot be changed to refer to another variable. If you use & operator on a reference, you will get the address of the variable it refers to.

For example, a swap function can be implemented as follows:

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

However, if you write the function like above, you lose the ability to auto-convert. But you can explicitly convert the variable to a reference. For example:

```
long a = 0;
long b = 1;
swap(a, b);                  // will trigger an error
swap((int &) a, (int &) b); // correct
```

If the argument is not a lvalue, you cannot pass by reference. For example:

```
void modifyReference(int &x) {
    x = 10;
}

int main() {
```

```
    modifyReference(5); // 5 is an rvalue, will trigger an error
}
```

But if you declare the parameter as a const reference, you can pass by rvalue. For example:

```
void print (const int &x) {
    cout << x << endl;
}

int main() {
    print(5); // correct
}
```

Or you can return a reference from a function. This saves the overhead of copying the return value. But you should not return a reference to a local variable, because the local variable will be destroyed when the function terminates. The more common use case is to return a reference to a parameter. For example:

```
// very wrong, c will be a dangling reference
int & add(int &a, int &b) {
    int c = a + b;
    return c;
}
```

6.5.1 Const Reference

You can use const reference to pass by value. This will prevent the function from modifying the original variable.

String type is special, because they are often treated as const char * when passed to a function. However, you can also use const string & to pass by value (this accepts both string and const char *). But do be careful if you pass a pointer to a single char to such function, because the function will treat it as a string. As the single char is not null-terminated, this will cause an error.

If the return type of a function is a reference, it is a valid lvalue. In the case that you don't want to modify the original variable, you can return a const reference.

6.6 Default Argument

You can specify default arguments for a function. If you do not pass a value to the argument, the default value will be used. A function can have multiple default arguments, but all the default arguments must be added from right to left (the actual argument assignment is from left to right).

If you separate the function declaration and definition, you can only specify default arguments in the declaration. If you did not specify default arguments in the declaration, the compiler will consider the function as no default arguments. If you provide default arguments both in the declaration and definition, you will get an error.

6.7 Function Overloading

Function overloading is a feature that allows us to have more than one function with the same name, as long as they have different parameters. The compiler will perform type conversion to determine which function to call.

To be noticed, most of the time, reference type and normal type are considered the same. But if a conversion is performed before passing the argument, the compiler will invoke the normal type function. This also works when you directly provide a value in the function call. For example:

```
void print(int &x) {
    cout << "int &" << endl;
}
void print(int x) {
    cout << "int" << endl;
}

int main() {
    long a = 0;
    print(a); // will print "int"
    print(1); // will print "int"
}
```

Also, function with pointer and const pointer are considered overloaded. If you do so, any pointer points to a const variable will go to the const pointer function, and the rest will go to the pointer function. For example:

```
void print(int *x) {
    cout << "int *" << endl;
}
void print(const int *x) {
    cout << "const int *" << endl;
}

int main() {
    int a = 0;
    const int b = 0;
    print(&a); // will print "int *"
    print(&b); // will print "const int *"
}
```

6.8 Function Template

A function template is a function that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type. For example, a function template for swapping two variables can be implemented as follows:

```
template <typename T>
void swap(T &a, T &b) {
```

```
    T temp = a;
    a = b;
    b = temp;
}
```

You can specialize a function template for a specific type. For example:

```
template <>
void swap<int>(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
```

A specialized template will override the original template. And a normal function will override both the template and the specialized template.

7 Duration, Scope, and Linkage

A typical structure for a C++ program consists of three parts: A header file, a source file, and a main file.

Common header files include:

- Function prototypes
- Constant definitions
- Structure declarations
- Class declarations
- Template declarations
- Inline function definitions

Remember to include header file in just one source file. If not sure, you can always use guards to prevent multiple inclusion.

Common source files include all other functions and class definitions. And the main file is where the main function is defined.

7.1 Storage

7.1.1 Automatic Storage

Automatic storage is the default storage class for all local variables. They are stored in stack, and will be destroyed when the function terminates.

All automatic variables have automatic storage, local scope, and no linkage.

7.1.2 Static Storage

Static storage is used to store variables that exist throughout the lifetime of the program.

A no linkage static storage variable is only visible within the block or function. The only way to declare a no linkage static storage variable is to declare a static variable inside a function. This will make the variable visible throughout the function, but not outside the function.

An external linkage static storage variable is visible throughout all files. When you declare a static variable outside any function, the variable will be visible throughout the file. However, in other files, you must declare the variable as `extern` to use it. Although the variable has static storage, it doesn't have to be declared as static (actually if you do so, it will become an internal linkage static storage variable).

An internal linkage static storage variable is visible throughout the file. You can declare the variable as `static` to make it an internal linkage static storage variable.

7.1.3 Dynamic Storage

Dynamic storage is used to store variables that are created by `new` operator. They are stored in heap, and will exist until you delete them.

7.2 Scope and Linkage

7.2.1 Scope

The scope of a variable is the part of the program where the variable can be directly accessed.

1. Local Scope: Only visible within the block.
2. Global Scope: Visible once declared until the end of the file.
3. Function Prototype Scope: Visible within the parentheses of the function prototype.
4. Class Scope
5. Namespace Scope

7.2.2 Linkage

The linkage of a variable is the part of the program where the variable can be indirectly accessed.

1. No Linkage: Only visible within the block or function.
2. Internal Linkage: Only in the same file.
3. External Linkage: Shared across multiple files.

7.3 About Function

All functions automatically have external linkage. If you want to make a function private to a file, you can declare it as `static`.

8 Namespace

A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Multiple namespace blocks with the same name are allowed.

A namesapce cannot be declared inside a block. All name declarations have external linkage by default.

A global variable (a variable that is not declared inside any function) is in the global namespace.

9 Class

9.1 Access Control

A class has three access specifiers: `public`, `private`, and `protected`. The default access specifier is `private`.

9.2 Inline Member Function

A member function defined inside a class declaration is implicitly an inline function. However, you can make a member function explicitly inline by using the `inline` keyword.

9.3 Constructor

A constructor is a special member function that is called when an object is created. It is used to initialize the object's data members.

A constructor has the same name as the class, and it does not have a return type. A default constructor is provided by the compiler if you do not define any constructor, which has no parameters and does nothing. To be noticed, to invoke the default constructor, you must not use parentheses (this will be considered as a function declaration).

Once you define a constructor, the compiler will not provide a default constructor.

9.4 Destructor

A destructor is a special member function that is called when an object is destroyed. It is used to free the object's resources. It takes no parameters and has no return type. Its name is the class name preceded by a tilde (`~`).

A destructor should free memory especially when you dynamically allocate memory in the constructor. Moreover, if you dynamically allocate memory in the constructor, you should also define a copy constructor and an assignment operator. If class inherits from another class, you should also define a virtual destructor.

9.5 Const Member Function

A const member function is a member function that promises not to modify the object.

9.6 This Pointer

The `this` pointer is a pointer that points to the object itself. Its most common use is to resolve name conflicts between class members and function parameters. Another use is to return a reference or a pointer to the object itself.

9.7 Enum Class

An enum class is a scoped enumeration that is strongly typed. The member names of an enum class cannot be implicitly converted to integers. To declare an enum class, you can use the following syntax:

```
enum class Color {
    RED,
    GREEN,
    BLUE
};
```

9.8 Const Member

You can declare a member variable as `const`. But this cannot be used to initialize other member variables. For example:

```
class A {
    const int size = 10;
    int a[size]; // will trigger an error
};
```

To assign value to a const member variable in function body is not allowed, just like any other const variable. If you want to initialize a const member variable, you can use the following syntax:

```
class A {
    const int size;
public:
    A(int size) : size(size) {}
};
```

You can use enum type or static const member variable to initialize other member variables. For example:

```
class A {
    enum {size = 10};
    int a[size];
};
class B {
    static const int size = 10;
    int a[size];
};
```

9.9 Static Member

A static member variable is a variable that is shared by all objects of the class. It must be initialized outside the class.

9.10 Const Member Function

A const member function is a member function that promises not to modify the object. It can only call other const member functions.

A const variable can only call const member functions.

9.11 Static Member Function

A static member function is a member function that is shared by all objects of the class. It can be called without an object, by using the class name and the scope resolution operator. But accessing it like a normal function is also allowed. For example:

```
class A {
    static int a;
public:
    static void print() {
        cout << a << endl;
    }
};
int A::a = 0;
int main() {
    A::print();
    A a;
    a.print();
}
```

9.12 Copy Constructor

A copy constructor is a constructor that creates an object by copying another object. It is used to initialize an object with another object of the same type. For example:

```
class A {
    int a;
public:
    A(int a) : a(a) {}
    A(const A &a) : a(a.a) {}
};
```

The copy constructor is provided by the compiler if you do not define any constructor. However, it just does a shallow copy, this will cause error if you dynamically allocate memory in the constructor. Because two pointers will point to the same memory location, and when one of them is destroyed, the other one will become a dangling pointer. Also, if you have a counter in the class, you should also define a copy constructor.

10 Operator Overloading

A class can overload most operators. However, some operators cannot be overloaded:

- `::` (scope resolution operator)
- `.` (member access operator)
- `.*` (member access through pointer to member operator)
- `?:` (ternary operator)
- `sizeof` (object size operator)
- `typeid` (object type operator)

Also, all castings cannot be overloaded:

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`

The overloaded operators can be called explicitly or implicitly. For example, the `operator+()` function can be called explicitly as `a.operator+(b)` or implicitly as `a + b`.

10.1 Overloading Binary Operators

Since the function call is always on the left, the left operand of a binary operator must be an object of the class. However, if you want to overload a binary operator with a built-in type on the left, you must declare the function as a friend function.

A friend function, although declared inside the class, is not a member function. This means any instance of the class cannot directly invoke the friend function, instead, the function is automatically called when the binary operator is used.

To be noticed that the order of the operands matters. If the order is reversed, the compiler will not be able to find the function.

10.2 Conversion

If you don't use the `explicit` keyword, constructor of the class can be used for implicit conversion. This is troublesome as you often don't know when the compiler will use the constructor.

To convert a class into another type, you can overload the cast operator. For example:

```
class A {
    int a;
public:
    A(int a) : a(a) {}
    operator int() {
        return a;
    }
};
```

Notice that there is no return type for the cast operator. You can also use the `explicit` keyword to prevent implicit conversion.

10.3 Assignment Operator

You need to overload the assignment operator if you dynamically allocate memory in the constructor. Different from a copy constructor, the assignment operator is called when the object already exists, hence there is no need to update the counter. Also, you need to delete the content of the object before assigning a new value to it. Remember to check if the object is the same as the one on the right side of the assignment operator.

11 Inheritance

When using inheritance, you can use relaxed access control. For pointers and references, a base class pointer or reference can point to a derived class object, but not vice versa. This is because the derived class object has more members than the base class object.

A pointer or reference used in this way has two behaviors: static type and dynamic type. If the function is not virtual, the pointer or reference will call the function of its type. If the function is virtual, the pointer or reference will call the function of the object's type.

11.1 Public Inheritance

Public inheritance shapes is-a relationship. In public inheritance, the public members of the base class become public members of the derived class. The protected members of the base class become protected members of the derived class. The private members of the base class are not accessible by the derived class.

11.2 Private Inheritance

Private inheritance shapes has-a relationship. All members of the base class become private members of the derived class.

If you use private inheritance, you must access the member through type name. For example:

```
class A : private std::string {
    A (const std::string &s) : std::string(s) {}
    void print() {
        std::cout << std::string::size() << std::endl;
    }
};
```

If you want to use the member itself, you need to cast the object to the base class. For example:

```
class A : private std::string {
    A (const std::string &s) : std::string(s) {}
    void print_content() {
```

```
        std::cout << (std::string) *this << std::endl; // or cast to a reference
    }
};
```

12 Class Template

A class template is a class that can operate with generic types. This allows us to create a class template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

13 Friend Class

A friend class is a class that can access the private and protected members of another class. To declare a friend class, you can use the following syntax:

```
class A {
    friend class B;
};
```

In this way, all members of class B can access the private and protected members of class A. But class A cannot access the private and protected members of class B.

Friend class is not symmetric, not transitive, and not inherited.

14 Exception

When an exception is thrown, the program will jump to the nearest catch block. If not, this is call the function `abort()` and terminate the program.

In C++, a catch clause can catch more than just exception, it can also catch any type.

In a catch chain, the matching will be executed from top to bottom. Hence, considering inheritance of exception, you should put the catch clause of the derived class before the catch clause of the base class.

15 RTTI

In C++, up-casting is safe and can be done implicitly. But down-casting is not safe. We have to use `dynamic_cast` to perform down-casting.

If you try to down-cast a pointer to a base class to a pointer to a derived class, and the pointer does not point to a derived class, the result will be a `nullptr`. If you try to down-cast a reference to a base class to a reference to a derived class, and the reference does not refer to a derived class, the result will be a `bad_cast` exception.

To find the true type of an object, you can use `typeid` operator. Remember if you want to apply this to a pointer, you need to dereference it first. Otherwise you will get the type of the pointer itself.

16 Lab Review

16.1 Scanf and Printf with Format Specifiers

Type	Format Specifier
short	%hd
int	%d
long	%ld
long long	%lld
float	%f
double	%lf
long double	%Lf
char	%c
string	%s
pointer	%p

Table 3: Scanf and Printf with Format Specifiers

If you want the unsigned version of the integer types, simply replace `d` with `u`.

To be noticed, the `sizeof` operator returns in long type.

16.2 IOS Manipulators

`setw` is used to set the width of the next output. You can use it with `setfill` to set the fill character.

`percision` is used to set the percision of the next output. This will round the number instead of truncating it.

Some other useful manipulators are:

- `hex`, `oct`, `dec`: change the base of the next output.
- `fixed`: output the number in fixed-point notation.
- `boolalpha`: output the boolean value as `true` or `false`.
- `showpoint`: always show the decimal point.
- `left`, `right`: align the output to the left or right.

16.3 String

When you use C-style string, there are two ways to get the length of the string:

1. Use `strlen` function (might not work if the string is not null-terminated).
2. Use `sizeof` operator (this will include the null character).

When you use `string` class, you can use `.size()` or `.length()` method to get the length of the string. However, using `sizeof` operator on a `string` object will return the size of the object (which is 32), not the length of the string.

It is particularly troublesome to input a string with spaces.

In C, we use `gets` and `puts` function. This works with a char array, and it does not include the newline character. To be noticed that it may input a string without a null character, if the input is the same length as the array.

In C++, we use `get` and `getline` function. When using `get` function, it requires two parameters: the first is a char array, and the second is the size of the array. This will always include a null character even if the input is longer than the array. You can use a `get` with no parameter to remove the newline character from the input stream (since `get` will stop at the newline character). When using `getline` function, the parameters are the same as `get` function, but it will remove the newline character from the input stream automatically.

```
char a[10];
cin.get(a, 10);
cin.get();          // remove the newline character
cin.getline(a, 10); // no need to remove the newline character
```

When dealing with string class object, we use `getline` a little differently.

```
string a;
getline(cin, a);
```

16.4 Union and Endianness

A union is a compound data type that allows you to store different data types in the same memory location. All members are aligned to the same memory location. But only one member can contain a value at any given time. The other members will be overwritten, and if you try to access them, their value will depend on the endianness of the computer.

Big-endian means that the most significant byte is stored at the lowest memory address. Little-endian means that the least significant byte is stored at the lowest memory address.

16.5 Pointer

If you apply `sizeof` operator on a pointer, it will return the size of a pointer (4 or 8), not the size of the array.

Notice that a char variable's address cannot be directly printed with & operator, you need to cast it to a `void*` first.

In C, there are four functions to dynamically allocate memory:

- `void* calloc(size_t num, size_t size)`: allocate memory for an array of `num` elements, each of them `size` bytes long, and initializes all bits to zero.
- `void* malloc(size_t size)`: allocate memory block of `size` bytes.
- `void* realloc(void* ptr, size_t size)`: reallocates memory extending it up to `size` bytes.
- `void free(void* ptr)`: deallocate the memory previously allocated by a call to `calloc`, `malloc`, or `realloc`.

In C++, it is recommended to use `new` and `delete` operator instead.

The name of an array is actually a pointer of one rank lower. For example, the name of a one dimensional array is a pointer, the name of a two dimensional array is a pointer to a one dimensional array, and so on. Here are some useful pointer arithmetic with array. For one dimensional array:

- `p + 1`: points to the second element of the array.
- `&p + 1`: points to the address after the array.
- `*p + 1`: equivalent to `p[0] + 1`.

For two dimensional array:

- `p + 1`: points to the second row of the array (remember `p` is a pointer to array and points to the first row).
- `*(p + 1)`: if you try to dereference `p + 1`, you will get the second row of the array. However, this decays to a pointer to int, which points to the first element of the second row.

When you mix pointer arithmetic with increment and decrement, you should be noticed that the dereference operator has a lower precedence than the increment and decrement operator. For example `*p++` is equivalent to `*(p++)`.