COMP 322 Introduction to C++

Junji Duan

Last updated: 2024/4/16

Contents

PS

Special thanks to Dr. Chad Zammar, the instructor of this course, who provided us with a joyful learning experience. Thanks to him, the class was much more enjoyable and engaging than one might expect. This note is prepared based on Dr. Zammar's lecture slides.

COMP 322 Course Outline

Junji Duan

2024/1/4

Goals of the course

By the conclusion of this course, you will have achieved the following:

- Discernment of Distinctions: Develop a comprehensive understanding of the distinctions between C++, Java, and C, enabling you to navigate and appreciate the unique characteristics of each language.
- Memory Management Proficiency: Acquire the skills to adeptly manage memory in a programming environment devoid of garbage collection, demonstrating competence in a crucial aspect of software development.
- Application of C++ Features: Gain practical knowledge in utilizing key features of C++, enhancing your proficiency in leveraging the capabilities of this versatile programming language.
- Foundational Understanding of OOP: Master the basics of object-oriented programming, a fundamental paradigm in software development, enabling you to design and implement efficient and modular code.
- Strategic Application of Techniques: Familiarize yourself with a spectrum of potent techniques within the realm of C++, and develop the discernment to apply them judiciously, elevating the quality and efficiency of your programming endeavors.
- Broadened Programming Perspective: Cultivate a more nuanced and enriched understanding of programming by witnessing alternative approaches to problem-solving, particularly contrasting with the paradigms employed

Textbook

- The C++ Programming Language by Bjarne Stroustrup.
- Another highly recommended books are:
  - Accelerated C++: practical programming by example by Andrew Koenig and Barbara Moo.
  - C++ primer by Stanley B. Lippman, Josée Lajoie and Barbara E.Moo

Evaluation

- There will be three homework assignments and two quizzes:
  - 3 assignments worth 20% each (total 60%)
  - 2 quizzes worth 20% each (total 40%)

Assignments must be submitted electronically via mycourses, by the due date. Late assignments will be penalized by 10 percent per day up to a maximum of 2 days. For example, an assignment received 25 hours late will be eligible for at most 80% of the possible score. An assignment received 50 hours late will not be accepted except in extenuating circumstances (e.g. illness).

Approximate Schedule

1. 05 Jan - Hello World, Course introduction and some basics of C++
2. 12 Jan - Flow control, Functions and Input/Output
3. 19 Jan - Pointers and references (Assignment 1 out)
4. 26 Jan - Memory management
5. 02 Feb - More on pointers, arrays and strings
6. 09 Feb - Classes in C++ (Assignment 1 due)
7. 16 Feb - Quiz 1 (Assignment 2 out)
8. 23 Feb - Classes and Inheritance
9. 01 Mar - Reading week
10. 08 Mar – Operator Overloading (Assignment 2 due, Assignment 3 out)
11. 15 Mar - Exceptions
12. 22 Mar - Templates
13. 29 Mar- std template libraries and a word about C++14, 17 and 20. (Assignment 3 due)
14. 05 Apr - Quiz 2

COMP 322 Lecture 1 - Introduction to C++

Junji Duan

2024/1/5

Today's Outline

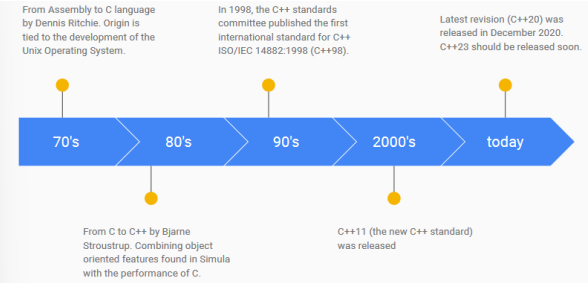
- Basic for C++
- C++ from C perspective
- C++ from Java perspective

Resources

- Recommended books:
  - The C++ Programming Language by Bjarne Stroustrup
  - Accelerated C++: Practical Programming by Example by Andrew Koenig
  - C++ Primer by Stanley Lippman and Barbara Moo
- C++ tutorials on the Internet (www.cplusplus.com, www.learncpp.com)

History of C++

- General purpose programming language
- Evolved from the C programming language
- Multi-paradigm language
- Low level memory manipulation
- Standard template library for data structures and algorithms



C++ impact and popularity

- Quickly gained popularity for system-level programming and application requiring performance
- Used in various domains
  - Game programming
  - System programming / Embedded systems
  - High-performance computing
  - Financial softwar

C++ today

- Remains one of the most widely used programming languages
- Known for its efficiency, flexibility, and a large ecosystem of libraries and frameworks
- Ongoing development with community contributions and standard updates
- Legacy code and projects written in C++ continue to be maintained and expanded

C++ vs C design

- C++ derives directly from C, so it is C plus plus more stuff
- Object Oriented via Classes

  - In C data and functions are separated
    In C++ they are encapsulated within a class where data can be hidden
- Generic programming via Templates
- Redesigned memory management via new/delete
- Operator (and functions) overloading
- Namespaces
- Reference variables
- Exception handling

C++ vs Java design

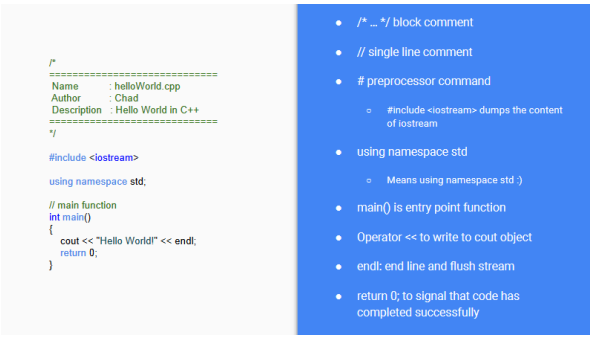
- C++
  - Compiled language
    - Runs as native binary on a target
  - Compatible with C code (very few rare exceptions)
  - Allows multiple programming paradigms without discrimination
  - Allows multiple inheritance of classes
  - Allows manual low level memory management via pointers
- Java
  - Interpreted language
    - Runs through a virtual machine
  - Uses JNI (Java Native Interface) to call C/C++ code
  - Allows multiple programming paradigms but strongly favors Object-Oriented
  - Single inheritance for classes
  - No pointers and provides automatic garbage collection mechanism

Compiling and running C++ code (1/2)

- C++ code can be contained
  - in a single file or
  - it can span over multiple files
  - **Common practice to separate header files (.h) containing declarations from implementation files (.cpp)**
  - Remember that, unlike C++, Java requires a different file for each public class and requires that the name of the file matches the name of the class
- C++ common file extensions:
  - .cpp, .c++, .c, .cxx, .cc, .hpp ···
- Use any file editor or IDE to write C++ code

Compiling and running C++ code (2/2)

- C++ code need to be compiled in order to run as an executable
- gcc / g++ under Linux, MSVC under windows, Clang under OSX
- Example: **g++ example.cpp -o example**
- Usually when we compile we invoke two operations (actually 3 if we consider pre-processing):
  - Compilation: transforms source code to object file (intermediate step between source code and final executable file)
  - Linking: producing one executable file from multiple object files
- Common practice is to use special script called Makefile for compiling complex projects
- www.cpp.sh : simple frontend gcc compiler for quick coding and debugging



Data Types (1/2)

- Same as C
  - int : sizeof(int) = 4 (4 at least, on 64-bit system)
  - float : sizeof(float) = 4 (usually 4, which is a 32-bit floating point type)

  - char : sizeof(char) = 1
  - double : sizeof(double) = 8 (which is a 64-bit floating point type)

Sizeof results may vary depending on compiler and operating system (32-bit vs 64-bit)

Data Types (2/2)

- C++ only
  - string : sizeof(string) = 8 in general on a 64-bit system
  - bool : sizeof(bool) = 1 in general but it is implementation dependent so might differ from 1
  - auto (since C++11): type automatically deduced from initializer
    * Do not confuse with C auto modifier which is the default for all local variables

Sizeof results may vary depending on compiler and operating system (32-bit vs 64-bit)

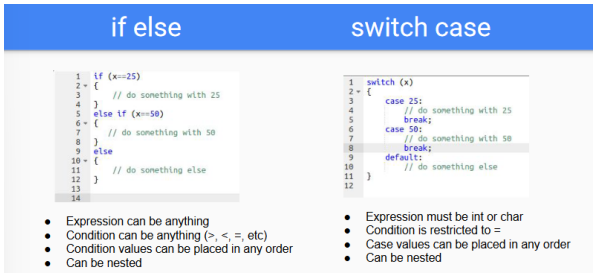
Operator Review

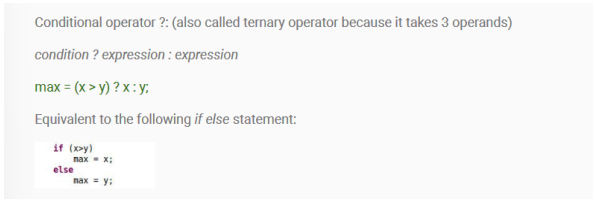
- Assignment operator (=)
  - To assign (from right to left) a value to a variable
  - int x = 42;
- Mathematical operators (+, -, *, /, %)
  - Arithmetical operations: add, subtract, multiply, divide, modulo
  - int x = 13%3;
- Relational operators (==, !=, <, <=, >, >=)
  - Test based on comparison
- Logical operators (&&, ||, !)
  - AND, OR, NOT
- Bitwise operators (&, |, ~, ^, «, »)
  - AND, OR, NOT, XOR, left shift, right shift

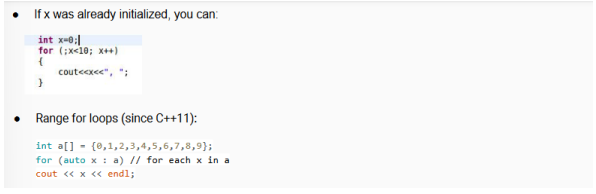
Flow control

- Conditional execution
  - if (condition) ···else ...
  - switch (expression) case constant ···
- Loops (iterate over the same code multiple times)
  - for (initialization; condition/termination; increment/decrement)
  - for (element:array)
  - while (condition) { ···}
  - do { ···} while (condition)

Relational or logical operators can be used to evaluate conditions







- What would the following for loop do?
  - for(; ;)
    * That would be Infinite loop

COMP 322 Lecture 2 - C++ Basics

Junji Duan

2024/1/12

Today's Outline

- Standard input & output
- Namespaces

Standard input/output

- C++ uses "streams" for reading from (input) and writing to (output) a media
  - Media can be a keyboard, screen, file, printer, etc.
- Input and output streams are provided by the iostream header file
  - **include <iostream>**
- cout stream object is used to print on screen
  - **cout « "some message";**
  - «: insertion operator
  - cout: object of ostream class
- Default standard output is the screen
- Similar to printf() in c, system.out.println() in java

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello";
    cout << "Class";
}
```
Output:
HelloClass

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello" << "Class";
}
```
Output:
HelloClass

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello" << endl << "Class";
}
```
Output:
Hello
Class

- cin stream object is used to read from the keyboard
  - cin » x;
  - »: extraction operator
  - cin: object of istream class
- Cin can read strings but limited to one word
  - cin » stringVariable;
- Use getline function to read a full sentence
  - getline(cin, stringVariable);
- Similar to scanf() in c, scanner class in java

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

int main()
{
    string var;
    cout << "Please enter your name" << endl;
    cin >> var;

    cout << "your name is: " << var;
}
```

Namespaces

- A name can represent only one variable within the same scope
- Large projects consists of multiple modules of code provided by different programmers
  - What happens if one module has a variable name that is the same as another variable in different module? **Name conflict (also called name collision)**
- Namespaces solve the name conflict problem

QuebecTemp.h
```
namespace QC
{
    double getTemp()
    {
        return -30.7;
    }
}
```

main.cpp
```
#include <iostream>
#include "QuebecTemp.h"

int main() {
    std::cout << "Temperature is: " << QC::getTemp() << std::endl;
    return 0;
}
```

Or also: main.cpp
```
#include <iostream>
#include "QuebecTemp.h"

using namespace QC;

int main() {
    std::cout << "Temperature is: " << getTemp() << std::endl;
    return 0;
}
```

COMP 322 Lecture 3 - C++ Basics

Junji Duan

2024/1/19

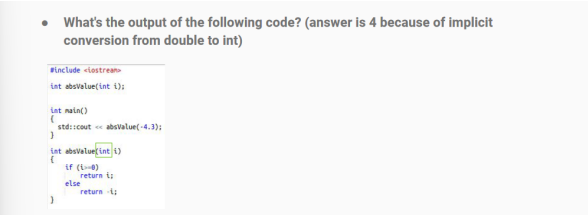
Today's Outline

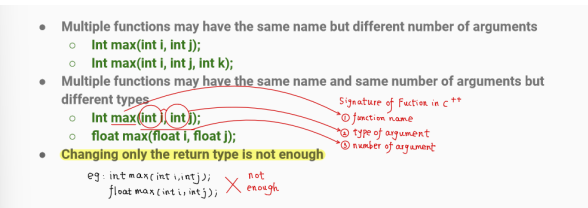
- Functions

Functions

- Same as in C and java
- Should be declared before being used
- Declaration should include the name, return type and arguments type
  - Also called prototype or signature of a function
- If the function doesn't return a value, its return type should be declared void
- Functions can be recursive

Functions Overloading





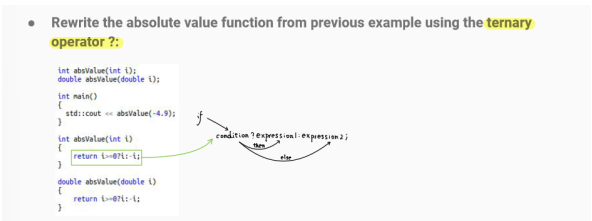
Function overloading is a feature in C++ that allows you to define multiple functions with the same name but different parameter lists. When you call an overloaded function, the compiler determines which function to call based on the provided parameter types or number.

```cpp
#include <iostream>

// Function overloading example
int add(int a, int b) {
    return a + b;
}

// Overloaded function
double add(double a, double b) {
    return a + b;
}

int main() {
    int sum_int = add(3, 4); // Calls the first add function
    double sum_double = add(3.5, 4.2); // Calls the second add function

    std::cout << "Sum of integers: " << sum_int << std::endl;
    std::cout << "Sum of doubles: " << sum_double << std::endl;

    return 0;
}
```

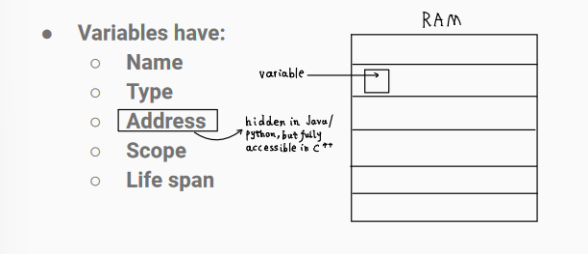
In this example, we define two functions named add, each taking different parameter types (one accepts integers, the other accepts double-precision floating-point numbers). When we call the add function in the main function, the compiler determines which version of the add function to call based on the provided parameter types.

Thus, even though the function names are the same, the compiler can distinguish between them due to different parameter lists and selects the appropriate function to call. This is the essence of function overloading in C++.

Small Quiz

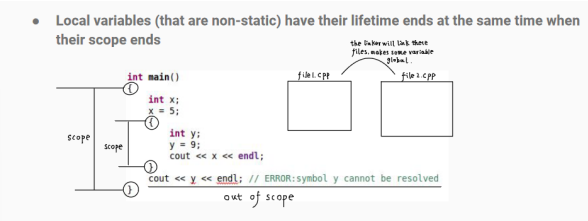


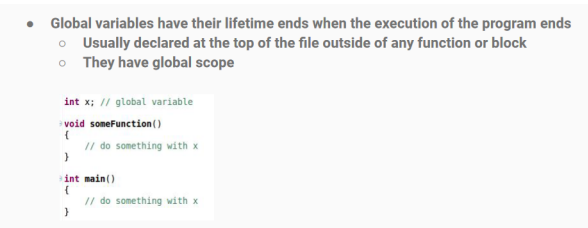
More about variables ...

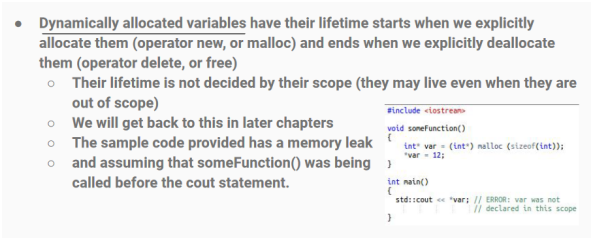


Scope and lifetime of a variable

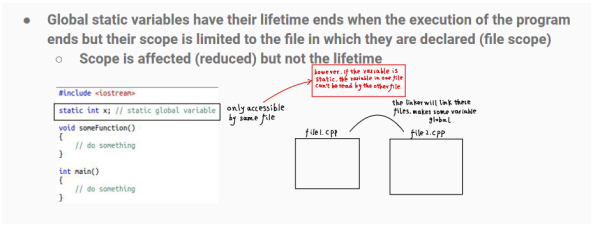
- When declaring variables we specify the name and type, but we should also keep in mind their scope and lifetime
- Scope of a variable
  - A section of the program where the variable is visible (accessible)
- Lifetime of a variable
  - The time span where the state of a variable is valid (meaning that the variable has a valid memory)
- Local variables (that are non-static) have their lifetime ends at the same time when their scope ends
  - Local variables may also be called automatic variables because they are automatically destroyed at the end of their scope
  - Scope of local variables is comprised from the moment they are declared until the end of the block or function where they reside (in other terms, until the execution hits a closing bracket })

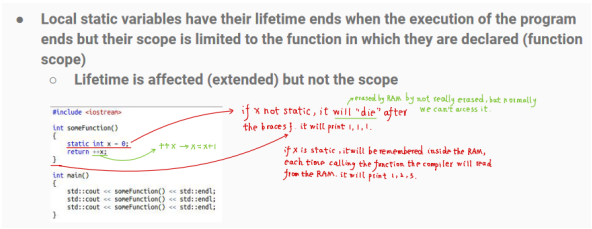






Scope and lifetime of a variable (static)





COMP 322 Lecture 4 - Pointers & References

Junji Duan

2024/1/26

Today's Outline

- Pointers
- Pointer arithmetics
- References

Pointers - Introduction

- Regular variables:
  - Are locations in memory
  - When declared, a memory address is automatically assigned
  - We don't care about their physical address
  - Accessible by their names
- Use the address operator & to get the physical address of a variable

```
int var;
cout << &var;
```

Output would be something similar to 0x7ffc8e229ddc

- To store the address of a memory location in a variable, we need a special type of variables called pointer variable
  - Use the dereference operator * to declare a pointer variable
  - int *ptr = &var;
    - ptr is a pointer variable
    - ptr stores the address of the variable var
    - ptr is pointing to var
    - ptr and &var are exactly the same thing
    - *ptr and var are exactly the same thing
    - The type of a pointer variable should match the type of the variable whose address is being stored: var is of type int, so *ptr should be of type int as well.

- Spaces don't matter when declaring pointers. The following declarations are all equivalent:
  - int *ptr;
  - int* ptr;
  - int * ptr;
- Be careful when declaring multiple variables on the same line
  - int * ptr1, ptr2; wrong
    - Only one pointer is being declared
    - ptr2 is not a pointer, it is simply an integer variable
  - int *ptr1, *ptr2;
    - Both variables are pointers

Pointers - code example

What's the output of the following code?

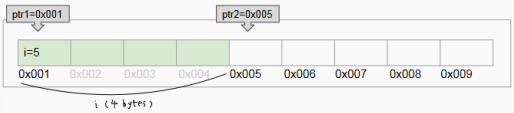
```
// main function
int main()
{
    int a = 7, b = 9;
    int *ptr1;
    int *ptr2 = &b;
    ptr1 = &a;

    *ptr1 = 15;
    *ptr2 = *ptr1;

    cout << a << '\n'; 15
    cout << b << '\n'; 15
    return 0;
}
```

Pointer Arithmetics

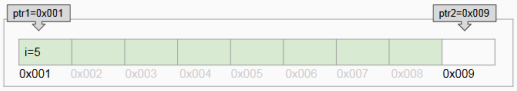
- Used to jump to different memory locations
- Only addition and subtraction (no multiplication and division)
- When you increment a pointer by one, it points to the next memory location
- Result depends on the size of the data type to which the pointer is pointing

- Imagine the memory as a table. Each cell is an element
- char *ptr; // ptr is a pointer of type char. Assuming that sizeof(char) = 1 byte
  - ++ptr; // will point to the next byte in the memory table (unless the compiler is applying memory padding or alignment for optimization)
- int *ptr; // ptr is a pointer of type int. Assuming that sizeof(int) = 4 bytes
  - ++ptr; // will point to a location that is 4 bytes away in the memory table

- int i = 5;
- int *ptr1 = &i; // ptr1 is a pointer of type int. Assuming that sizeof(int) = 4 bytes
- int *ptr2 = ++ptr1; // will point to a location that is 4 bytes away in the memory table



- double i = 5;
- double *ptr1 = &i; // ptr1 is a pointer of type double. Assuming that sizeof(double) = 8 bytes
- double *ptr2 = ++ptr1; // will point to a location that is 8 bytes away in the memory table



Pointers - Example 1

```
int main()
{
    int value = 100;
    int* pValue = &value;
    cout << "Value is equal to: " << *pValue << endl;
    cout << "Address of value = " << pValue << endl;
}
```

The output is:

```
Value is equal to: 100
Address of value = 0x7fff69d9b6b8
```

Pointers - Common mistakes

- Dereferencing invalid pointers
  - Uninitialized pointers point to random memory location

```
int main()
{
    int *ptr; // non-initialized pointer
              // points to random memory location
    *ptr = 12; // memory corruption

}
```

- Good practice to always:
  - assign pointers to NULL (or nullptr since Cx11) when they point to nothing
  - check if the pointer is not null before dereferencing it

```
int main()
{
    int *ptr = NULL;
    if (ptr != NULL)
    {
        cout << *ptr << endl;
    }
    else
    {
        cout << "pointer is NULL" << endl;
    }
}
```

- Mixing operator precedence rules to accidentally apply arithmetics on pointers instead of the value being pointed to
  - *++ptr; vs ++*ptr; // remember that ++ has higher precedence than *

Pointers - Example 2

```
int main()
{
    int value = 100;
    int* pValue = &value;
    cout << "Value is equal to: " << *pValue << endl;
    cout << "Address of value = " << pValue << endl;
    cout << ++*pValue << endl; // dereference the pointer
                               // then increment its value
    cout << *++pValue << endl; // increment the address of value
                               // then dereference the pointer
    cout << ++pValue << endl;  // increment the address of value
                               // to point to next memory location
}
```

The output is:

```
Value is equal to: 100
Address of value = 0x7ffcf1de7f34
101
-237076680
0x7ffcf1de7f3c
```

Pointers - Example 3

- What's wrong with the following function?

```
float *getPricePointer()
{
    float price = 9.99;
    return &price;
}
```

- What's wrong with the following function?
  - getPricePointer is returning the address of a local variable.
  - Local variables have limited scope and lifetime
  - price will be automatically destroyed as soon as the function returns
  - Its address will be pointing to an invalid memory location

```
float *getPricePointer()
{
    float price = 9.99;
    return &price;
}
```

Reference variables

- Reference variable is a C++ concept that doesn't exist in C
- Reference permits to assign multiple names to the same variable
- To declare a reference variable we use the address & operator
- int x;
- int &y = x; // be careful not to confuse with int *y = &x;
  - y is a reference to x
  - y is considered to be an alias for x
  - y and x are the same thing
  - y and x are two names for the same memory location

```
int main()
{
    int a = 100;
    int &b = a;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    b = 12;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;

}
```

The output is:

```
a = 100
b = 100
a = 12
b = 12
```

Passing arguments by reference

By Value

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

int getProduct(int x, int y)
{
    return x*y;
}

// main function
int main()
{
      int a = 4;
      int b = 5;
      int product = getProduct(a, b);
      cout << product;
}
```

By Reference

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

int getProduct(int &x, int &y)
{
    return x*y;
}

// main function
int main()
{
      int a = 4;
      int b = 5;
      int product = getProduct(a, b);
      cout << product;
}
```

By Pointer

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

int getProduct(int* x, int* y)
{
    return (*x)*(*y);
}

// main function
int main()
{
    int a = 4;
    int b = 5;
    int product = getProduct(&a, &b);
    cout << product;
}
```

By Reference

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

int getProduct(int &x, int &y)
{
    return x*y;
}

// main function
int main()
{
    int a = 4;
    int b = 5;
    int product = getProduct(a, b);
    cout << product;
}
```

Reference variables vs pointers

- Reference variables must be initialized
  - int &x = var;
- Reference variable cannot be changed to reference another variable
  - Similar to constant pointers
- Unlike pointers, references cannot be NULL
- Pointer has its own memory address whereas a reference shares the same memory address with the variable it is referencing
- Reference variables are very commonly used as function parameters
  - Better performance by avoiding copying values
- References are safer than pointers so they are preferred to pointers whenever you have the choice (if there is no need for dynamic allocation)

Pointers - are they worth the headache?

- **Pointers are used for**
  - **Efficiency (no need to statically reserve a huge array in advance)**
  - **Implementation of complex data structures**
  - **Dynamic allocation of memory**
  - **Passing functions as parameters**
  - **Many advanced C++ techniques**
- **Misusing pointers is the mother of all software bugs**
  - **Memory leaks**
  - **Dangling pointers**
  - **Buffer overflow**
  - **Abduction by aliens ... :)**

Pointers - confusing the cat (C++ interview question)

- **What's the output of the following code: 4, 5**

```
int main()
{
    int *ptr = NULL;
    int i = 7;  i++;
    for(int j=0; j<=2; j++) {
        i = j;
    }
    ptr = &i;
    if (ptr != NULL) {
        (*ptr) *= (*ptr);
    }

    if (ptr != NULL) {
        cout << (*ptr)++ << endl;
        cout << i << endl;
    }
    else {
        cout << "pointer is NULL" << endl;
    }
}
```

COMP 322 Lecture 5 - Memory Management

Junji Duan

2024/2/2

Today's Outline

- Automatic memory
- Dynamic memory
- Cleaning the mess
- Arrays vs pointers

Automatic memory

- Amount of memory to be allocated should be known in advance at compile time
- Any "normal" variable declaration uses automatic memory allocation (in some textbooks they refer to it as static memory. Do not confuse it with the static keyword. In this context they mean automatic memory)
  - int var;
  - int myArray[10];

  Compile time vs run time
- Automatic memory is automatically liberated when the variable goes out of scope
  - No need for programmer's intervention
- Automatic memory is being allocated from the "*Stack*" (fancy name for a region in memory)

Automatic memory - limitations

```
 9 // main function
10 int main()
11 {
12     int x = 5;
13     // some code affecting x
14     if (x <= 22)
15     {
16         int y = 12;
17         // do something ...
18     }
19     cout << y; // ??
20 }
```

This code won't compile ...

- Automatic variables cannot survive beyond their scope
- To be able to use the variable y, it should be declared global
  - What if y's creation is decided at runtime and not during compile time?

```
 9 // main function
10 int main()
11 {
12     int *x;
13     int a = 12;
14     if (a <= 22)
15     {
16         int y = 13;
17         x = &y;
18     }
19     cout << *x; // ??
20 }
```

- Using pointers to gain access to y's memory location will lead to undefined behavior
- This code does compile and run, however, there is no guarantee for what the value of *x would be

- What if the size of the memory needed isn't known until runtime?
  - Example: size of an array depends on user's input
- What if we needed a function or a block of code to return a pointer to a valid memory that was being declared within the function or block (similar to previous example)
- What if we have limited memory due to system constraints?
  - We cannot book in advance too much memory and keep it reserved for all the lifetime of the program

Dynamic memory

- Dynamic memory allocation does not require the amount of memory needed to be known before run time
- Dynamic memory is being allocated from the "*Heap*" (fancy name for a region in memory also known as the free store)
- Memory management is the responsibility of the programmer
  - Great power requires great wisdom
  - No Java style garbage collection

- Use **new** operator to dynamically allocate memory
  - Similar to **malloc** in C language

  malloc → memory allocation
- Use **delete** operator to free the allocated memory when it is not needed anymore
  - Similar to **free** in C language
- Use **new[ ]** operator to dynamically allocate a sequence of memory locations
  - Example: to allocate an array of elements
- Use **delete [ ]** operator to dynamically free a sequence of memory locations
- **delete** only memory allocated with **new**
- **delete [ ]** only memory allocated with **new [ ]**

C++ Style memory allocation

```
 9 // main function
10 int main()
11 {
12     int *x = new int;
13     *x = 5;
14     // do something with x
15     cout << *x;
16     delete x;
17 }
```

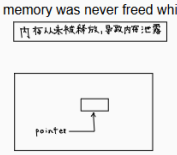
C Style memory allocation

```
 6 // main function
 7 int main()
 8 {
 9     int *x = (int*)malloc(sizeof(int));
10     *x = 5;
11     //do something with x
12     cout<<*x;
13     free(x);
14 }
```

- **Note that, unlike malloc, operator new is type-aware so no need for explicit type casting**

What's wrong with this code? (the memory was never freed which will lead to memory leak)

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

int* getIntPointer()
{
    int *ptr = new int;
    return ptr;
}

// main function
int main()
{
    int *x = getIntPointer();
    *x = 5;
    cout << *x;
}
```



need delete

- Arrays can be compared to constant pointers

```
// main function
int main()
{
    int array[3] = {3, 5, 7};
    int *ptr;
    ptr = array; // similar to ptr = &array[0]
    cout << ptr[1] <<endl;

    int array2[3] = {3, 5, 7};
    array2 = ptr; // this will not compile
}
```

array is constant pointer, so array can't point to something else.

Dynamic memory - pointer / array notation

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

int* getIntArray(int size)
{
    int *ptr = new int[size];
    return ptr;
}

// main function
int main()
{
    int *x = getIntArray(5);

    for(int i=0; i<5; i++)
    {
        *(x+i) = i;
    }

    for(int i=0; i<5; i++)
    {
        cout<<*(x+i)<<endl;
    }

    delete [] x;
}
```

new int; 1 element
new int[10]; 10 elements

Comparison between arrays and pointers

- Size of array should be predetermined in advance and cannot be resized after declaration
- The memory location of an array is fixed and cannot be changed after its declaration
- C++ supports other types of more robust arrays that we will discuss later (vector, map, etc.)
- Pointers can point to a chunk of memory of any size and this can also change during execution
- Unless declared constant, pointers can point to a different memory location later on during execution

Dynamic memory - initialization

- Operator new can initialize the newly created objects
  - **int *i = new int (2); // creates one element and initializes it to 2**
  - Same as: [] for number of element
    - **int *i = new int;**
    - ***i = 2;**
  - Do not confuse with:
    - **int *i = new int [2]; // creates an array of 2 uninitialized elements**
    - **int *i = new int [2](); // creates an array of 2 elements initialized to 0**
    - **int *i = new int [2]{3,5}; // creates an array of 2 elements initialized to 3 and 5. This is valid since C++11.**

Relationship between arrays and pointers

- Arrays can be compared to constant pointers

```
// main function
int main()
{
    int x = 5;
    int *var = &x;
    cout << var << endl;
```

Output is:

- 0x7ffee9c5c934

```
// main function
int main()
{
    int x[3] = {3, 5, 7};
    cout << x << endl;
    cout << &x[0] << endl;
    cout << &x[1] << endl;
}
```

address of array is the address of first element of the array.

Output is:

- 0x7ffe2caee980
- 0x7ffe2caee980
- 0x7ffe2caee984

- Arrays can be compared to constant pointers

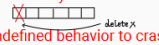
```
// main function
int main()
{
    int array[3] = {3, 5, 7};
    int *ptr;
    ptr = array; // similar to ptr = &array[0]
    cout << ptr[1] <<endl;
}
```

- Output: 5

Common mistakes

- "Writing in C++ is like running a chainsaw with all the safety guards removed" — by Bob Gray; cited in *Byte* (1998) Vol 23, Nr 1-4. p. 70
- "In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg" — by Bjarne Stroustrup the creator of C++

- Using delete to free memory allocated by new [] → memory leak
- Forgetting to free memory → memory leak



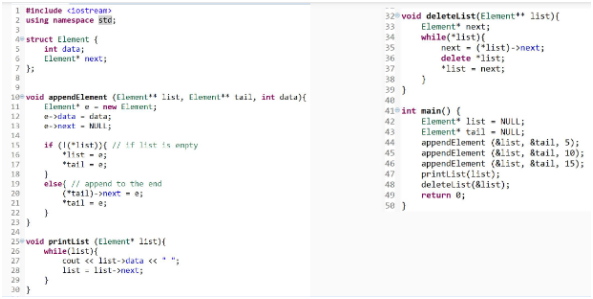
- Dereferencing a pointer after deleting it → from undefined behavior to crash
- Deleting the same pointer more than once → This will cause undefined behavior and weird crashes from the 5th dimension :)

```
// main function
int main()
{
    int *x = new int;
    int *y = x;
    *y = 7;
    cout<<*x;
    delete x;
    delete y;
}
```

C++11 Smart Pointers

- Note that since C++11 the standard library added more ways to manage dynamic memory in a safe way
  - auto_ptr // not deprecated but use unique_ptr instead
  - shared_ptr
  - unique_ptr // enhancement of auto_ptr
  - weak_ptr
- Wrappers to manage the lifetime of dynamically created objects and provide a garbage collection like environment
- We will get back to this after we cover Classes in C++

Pointer to Pointer - Linked List Example



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

struct Element {
    int data;
    Element* next;
};


void appendElement (Element** list, Element** tail, int data){
    Element* e = new Element;
    e->data = data;
    e->next = NULL;

    if (!(*list)){ // if list is empty
        *list = e;
        *tail = e;
    }
    else{ // append to the end
        (*tail)->next = e;
        *tail = e;
    }
}

void printList (Element* list){
    while(list){
        cout << list->data << " ";
        list = list->next;
    }
}
```



```
void deleteList(Element** list){
    Element* next;
    while(*list){
        next = (*list)->next;
        delete *list;
        *list = next;
    }
}

int main() {
    Element* list = NULL;
    Element* tail = NULL;
    appendElement (&list, &tail, 5);
    appendElement (&list, &tail, 10);
    appendElement (&list, &tail, 15);
    printList(list);
    deleteList(&list);
    return 0;
}
```

COMP 322 Lecture 6 - Classes in C++

Junji Duan

2024/2/9

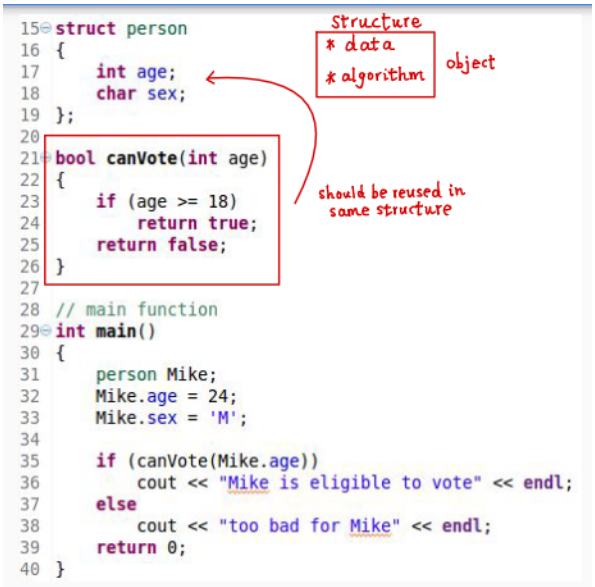
Today's Outline

- Review of OO concept
- C structs
- C++ Classes
- Constructors
- Destructors

OO programming: Quick review

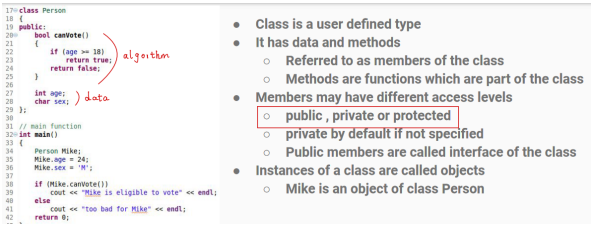
- Approach to design modular and reusable systems
- Programming is about manipulating data through code (methods or algorithms)
  - Object: is a coupling of both data and methods
- Extension to the concept of structures
  - Not only we group multiple data elements but we also attach the intelligence needed to manipulate the data (also called encapsulation)

C structs: data and code are separated

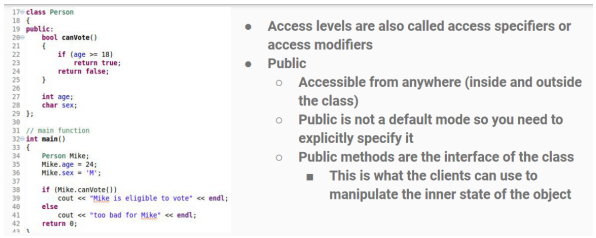



```
15 struct person
16 {
17     int age;
18     char sex;
19 };
20
21 bool canVote(int age)
22 {
23     if (age >= 18)
24         return true;
25     return false;
26 }
27
28 // main function
29 int main()
30 {
31     person Mike;
32     Mike.age = 24;
33     Mike.sex = 'M';
34
35     if (canVote(Mike.age))
36         cout << "Mike is eligible to vote" << endl;
37     else
38         cout << "too bad for Mike" << endl;
39     return 0;
40 }
```

C++ Classes: introduction

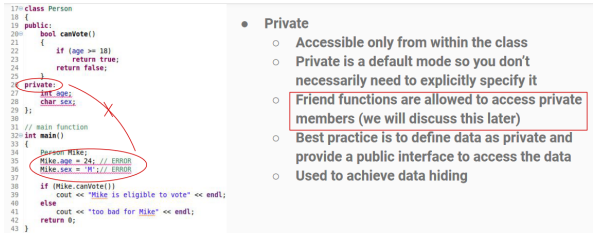


C++ Classes: access levels (public)



- Access levels are also called access specifiers or access modifiers
- Public
  - Accessible from anywhere (inside and outside the class)
  - Public is not a default mode so you need to explicitly specify it
  - Public methods are the interface of the class
    - This is what the clients can use to manipulate the inner state of the object

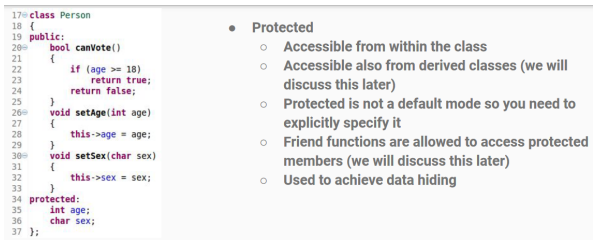
C++ Classes: access levels (private)



- Private
  - Accessible only from within the class
  - Private is a default mode so you don't necessarily need to explicitly specify it
  - Friend functions are allowed to access private members (we will discuss this later)
  - Best practice is to define data as private and provide a public interface to access the data
  - Used to achieve data hiding

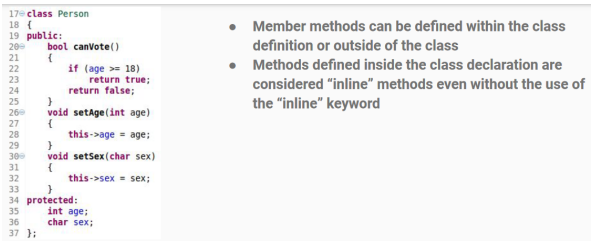
C++ Classes: access levels (private)

C++ Classes: access levels (protected)

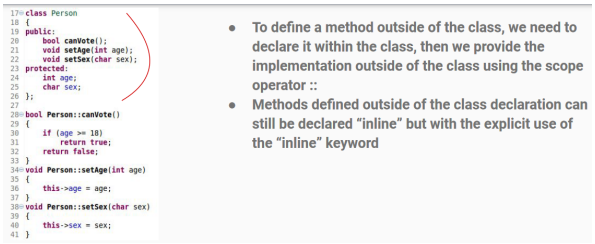


- Protected
  - Accessible from within the class
  - Accessible also from derived classes (we will discuss this later)
  - Protected is not a default mode so you need to explicitly specify it
  - Friend functions are allowed to access protected members (we will discuss this later)
  - Used to achieve data hiding

C++ Classes: method definition



- Member methods can be defined within the class definition or outside of the class
- Methods defined inside the class declaration are considered "inline" methods even without the use of the "inline" keyword



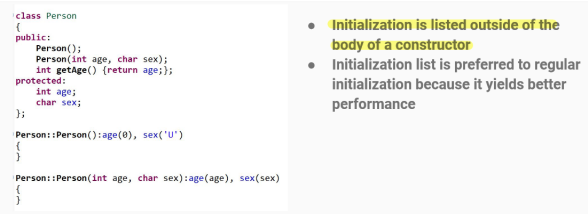
- To define a method outside of the class, we need to declare it within the class, then we provide the implementation outside of the class using the scope operator ::
- Methods defined outside of the class declaration can still be declared "inline" but with the explicit use of the "inline" keyword

C++ classes: constructors

- When instantiating a class, a special method is implicitly called first
  - This method is the constructor
- Every class has a constructor (at least one)
- If a constructor is not provided by the programmer, the compiler will provide a default implicit constructor (that does basically nothing)
  - This is how the construction of the Person class from the previous example was possible
- The Constructor method:
  - is used to initialize the data members of a class
  - must have the same name as the class
  - must be declared public in general
    - There are some exceptions when implementing advanced design patterns
  - does not have a return type
    - Constructors don't return values

- Constructor can be personalized using parameters
- User can define as many different constructors as needed

Constructor: initialization list



```
class Person
{
public:
    Person();
    Person(int age, char sex);
    int getAge() {return age;};
protected:
    int age;
    char sex;
};

Person::Person():age(0), sex('U')
{
}

Person::Person(int age, char sex):age(age), sex(sex)
{
}
```

- Initialization is listed outside of the body of a constructor
- Initialization list is preferred to regular initialization because it yields better performance

Public, Private and Protected

In C++, public, private, and protected are three types of access specifiers used to set the access level for class members. Below is an explanation of each access specifier and its usage through an example.

- public
  - Public members can be accessed anywhere, including outside the class. This means that if a class member is declared as public, any external code can access it directly.
- private
  - Private members can only be accessed by member functions of the same class. This means that if you attempt to access a private member from outside the class, the compiler will throw an error.
- protected
  - Protected members are similar to private members in that they restrict external access. However, unlike private, protected members are accessible in derived classes (subclasses).

Suppose we have a class named Person with three members: name, age, and height. We want name to be accessible by anyone, height to be accessible only within the class or its subclasses, and age to be accessible only within the class.

```
class Person {
public:
    string name;  // 公有成员，任何地方都能访问

protected:
    int height;  // 保护成员，只能在类内部及其子类中访问

private:
    int age;  // 私有成员，只能在类内部访问

public:
    Person(string n, int h, int a) : name(n), height(h), age(a) {}

    void printInfo() {
        cout << "姓名：" << name << ", 身高：" << height << ", 年龄：" << age << endl;
    }
};

class Student : public Person {
public:
    Student(string n, int h, int a) : Person(n, h, a) {}

    void printHeight() {
        cout << "学生身高：" << height << endl;  // 可以访问protected成员
    }
};
```

```
int main() {
    Person p("张三", 170, 30);
    p.printInfo();
    // cout << p.age;  // 错误：age是私有成员，不能在类外访问
    cout << "姓名：" << p.name << endl;  // 正确：name是公有成员

    Student s("李四", 160, 20);
    s.printHeight();  // 正确：派生类可以访问protected成员
    // cout << s.height;  // 错误：height在派生类外部不可访问
}
```

In this example, name is a public member, so it can be accessed directly in the main function. height is a protected member, so it can be accessed in the member function printHeight of the Student class (derived from Person), but not directly in the main function. Finally, age is a private member, so it cannot be accessed directly in the main function or in the Student class.

Struct, Class

In C++, struct and class are both composite data types used for data encapsulation, with the main difference lying in their default access level and default inheritance mode. In C++, types defined using struct or class can contain data members (member variables) and function members (member functions or methods), thereby encapsulating data and behavior.

struct

- Default access level: The members of a struct are public by default, meaning that by default, members of a struct can be accessed from outside the struct.
- Usage: Typically used for smaller data structures, more focused on storing data rather than emphasizing behavior.

```cpp
struct Person {
    std::string name;
    int age;

    void printInfo() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Person p1;
    p1.name = "张三";
    p1.age = 30;
    p1.printInfo(); // 直接访问并使用Person的数据成员和成员函数

    return 0;
}
```

class

- Default access level: The members of a class are private by default, meaning that by default, members of a class cannot be accessed from outside the class, ensuring encapsulation and data safety.
- Usage: Usually used to define complex data types containing both data and function members, more focused on the behavior and interface of objects.

```cpp
class Animal {
private:
    std::string name;
    int age;

public:
    Animal(const std::string& n, int a) : name(n), age(a) {}

    void printInfo() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Animal a1("小白", 5);
    a1.printInfo(); // 使用Animal的公有成员函数

    return 0;
}
```

Summary

- The essential difference between struct and class lies in their default access level (struct is public, class is private).
- In C++, both struct and class can be used to define types containing data members and member functions.
- The choice between struct and class often depends on your requirements for the default access level of the type and how you intend to use it (more emphasis on data or behavior).

P.S. Constructor and Destructor notes are in Lecture 7.

COMP 322 Lecture 7 - Classes in C++ 2

Junji Duan

2024/2/16

Today's Outline

- Constructors
- Destructors

C++ classes: destructors

- When an object gets out of scope, a special method is implicitly called
  - This method is the destructor
- Every class has a destructor (and only one)
- If a destructor is not provided by the programmer, the compiler will provide a default implicit destructor (that usually calls the destructor for each data member but will not delete dynamically allocated memory for you)
- Destructor method:
  - is used to clean and liberate any resource that was being held by the object
  - must have the same name as the class preceded by the tilde ~ operator
  - must be declared public in general
    - There are some exceptions when implementing advanced design patterns
  - does not have a return type nor does it take parameters
    - Destructors don't return values

```
class Person
{
public:
    // constructors
    Person();
    Person(int age, char sex);
    Person(int age, char sex, char *name);
    // destructor
    ~Person();
    int  getAge();
    char* getName();
protected:
    int age;
    char sex;
    char *name;
};
```

```
Person::Person(int age, char sex, char *name)
{
    cout << "Constructor got called" << endl;
    this->age = age;
    this->sex = sex;
    this->name = new char[strlen(name)];
    strcpy(this->name, name);
}

Person::~Person()
{
    cout << "Destructor got called" << endl;
    delete [] this->name;
}

char* Person::getName()
{
    return this->name;
}
```

```
Person::Person(int age, char sex, char *name)
{
    cout << "Constructor got called" << endl;
    this->age = age;
    this->sex = sex;     allocate 7
    this->name = new char[strlen(name)];
    strcpy(this->name, name);
}

Person::~Person()
{
    cout << "Destructor got called" << endl;
    delete [] this->name;
}

char* Person::getName()
{
    return this->name;
}
```

```
// main function
int main()
{
    Person Mike(24, 'M', "Michael");

    cout << Mike.getName() << endl;
}
```

actual output

```
Constructor got called
Michael
Destructor got called
```

C++ classes: dynamic allocation

```
class GPS
{
public:
    GPS(double altitude, double longitude, double latitude):
        altitude(altitude),
        longitude(longitude),
        latitude(latitude)
    {
        cout << "GPS Constructor" << endl;
    }

    ~GPS()
    {
        cout << "GPS Destructor" << endl;
    }

private:
    double altitude;
    double longitude;
    double latitude;
};

int main()
{
    cout << "Program started ..." << endl;
    GPS* gps;
    cout << "A GPS pointer was being declared but not allocated yet" << endl;
    gps = new GPS(10, 25, 14);
    cout << "GPS Object was being allocated" << endl;
    delete gps;
    cout << "GPS Object was being deleted" << endl;
}
```

```
Program started ...
A GPS pointer was being declared but not allocated yet
GPS Constructor
GPS Object was being allocated
GPS Destructor
GPS Object was being deleted
```

Classes - design pattern example

```
class Singleton
{
public:
    static Singleton& getUniqueInstance()
    {
        static Singleton instance;
        return instance;
    }
    // Add the needed public methods
    void doSomething() {}

private:
    Singleton(){}
    ~Singleton(){}
    Singleton(Singleton &);
    Singleton operator= (Singleton &);
};

int main()
{
    Singleton& mySingleton = Singleton::getUniqueInstance();
    mySingleton.doSomething();
}
```

- The Singleton design pattern provides an example of a case where declaring a constructor private makes sense
- Singleton enforces that only one object of a class can be present during the lifetime of a program

In C++, constructor and destructor are two special member functions that are automatically called at different stages of an object's lifecycle, used for object initialization and cleanup.

Constructor

- Definition: Constructor is a special member function used for initializing objects when they are created. It has the same name as the class and no return type.

- Purpose: Used for resource allocation, initializing member variables, etc.
- Types: Can have default constructor (no parameters), parameterized constructor, copy constructor, etc.
- Characteristics: Can be overloaded but cannot be inherited.

```cpp
class Student {
public:
    // 默认构造函数
    Student() {
        name = "未知";
        age = 0;
    }

    // 参数化构造函数
    Student(string n, int a) {
        name = n;
        age = a;
    }

private:
    string name;
    int age;
};
```

In the example above, the Student class has two constructors: one is the default constructor, called when an object is created without providing any parameters; the other is the parameterized constructor, used to initialize the member variables name and age when creating an object.

Destructor

- Definition: Destructor is a special member function used for performing cleanup tasks before an object is destroyed. It is named by prefixing the class name with a tilde ( ), has no return type, and takes no parameters.
- Purpose: Used for releasing resources allocated during the object's lifetime, such as dynamically allocated memory, file handles, etc.
- Characteristics: A class can have only one destructor, and it cannot be overloaded or inherited.

```cpp
class Student {
public:
    // 构造函数
    Student() {
        name = new string;
        *name = "未知";
    }

    // 析构函数
    ~Student() {
        delete name; // 释放动态分配的内存
    }

private:
    string* name;
};
```

In this example, the constructor of the Student class dynamically allocates a string to store the student's name. Its destructor ensures that this memory is released when the object's lifetime ends, avoiding memory leaks.

In summary, constructor and destructor are essential concepts in C++, responsible for object initialization and cleanup tasks, ensuring effective resource management.

COMP 322 Lecture 8 - Classes and inheritance

Junji Duan

2024/3/1

Today's Outline

- Friendship
- Inheritance
- Construction/Destruction order
- Types of inheritance
- Is-a VS Has-a
- Virtual methods
- Abstract classes

Classes - Behind the scenes

- How many methods does the following class have?

```
class SomeAwesomeClass
{

};
```

- Prior to C++11, the compiler would provide 4 methods for you unless you explicitly define them yourself:
  - Default constructor
  - Default destructor
  - Copy constructor
  - Copy assignment operator
- Since C++11, compiler will also generate 2 extra methods (so total now is 6):
  - Move constructor
  - Move assignment operator
- Probably other methods were being added in C++20

- Default constructor
- Default destructor
- Copy constructor
- Copy assignment operator

```
class SomeAwesomeClass
{

};

int main()
{
    SomeAwesomeClass sac1;
    SomeAwesomeClass sac2 = sac1;
    SomeAwesomeClass sac3(sac2);
    SomeAwesomeClass sac4;
    sac4 = sac1;
}
```

Classes - Copy Constructor

- SomeAwesomeClass(const SomeAwesomeClass & obj);
  - Instantiate and Initialize an object from another object having the same type
  - In Java we can obtain similar behavior by simply inheriting from "Cloneable" (however the way Cloneable works is very different from C++ copy constructor)

```
class SomeAwesomeClass
{

};

int main()
{
    SomeAwesomeClass sac1;
    SomeAwesomeClass sac2 = sac1;
    SomeAwesomeClass sac3(sac2);

}
```

Classes - Copy Assignment Operator

- SomeAwesomeClass & operator= (const SomeAwesomeClass & obj);
  - Assign an object from another object having the same type

```
class SomeAwesomeClass
{

};

int main()
{
    SomeAwesomeClass sac1;
    SomeAwesomeClass sac2 = sac1;
    SomeAwesomeClass sac3(sac2);
    SomeAwesomeClass sac4;
    sac4 = sac1;
}
```

Classes - friends

```
class GPS
{
public:
    GPS(double altitude, double longitude, double latitude):
        altitude(altitude),
        longitude(longitude),
        latitude(latitude)
    {
        cout << "GPS Constructor" << endl;
    }

    ~GPS()
    {
        cout << "GPS Destructor" << endl;
    }

    friend void setLongitude(GPS& gps);

private:
    double altitude;
    double longitude;
    double latitude;
};

void setLongitude(GPS& gps)
{
    gps.longitude = 42;
}
```

- Functions and classes can be declared "friends" using the friend keyword
- A friend function or class can have access to a class's private and protected members

What is class inheritance?

- Capability of a class to inherit (or extend) the members (data and methods) of another class
- Reuse of functionalities and characteristics of a base class by a derived class
- Multiple classes can derive from the same base class
- One class may derive from multiple base classes (unlike Java)

- Derived classes inherit all the accessible members of their base classes: public and protected members
- Derived classes can extend the inherited members by adding their own members
- Base class cannot access extended members defined within inherited classes

Class inheritance: example

```
class Aircraft
{
public:
    Aircraft() {cout << "Aircraft ctor" << endl;}
    ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}
    void fly() {cout << "Aircraft flying: " << capacity << endl;}
    // ...
protected:
    int capacity; //nbre of pass.
};

class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Boeing ctor" << endl;}
    ~Boeing(){cout << "Boeing ~dtor" << endl;}
};
```

```
// main function
int main()
{
    Aircraft a;
    a.setCapacity(50);
    a.fly();

    Boeing b;
    b.setCapacity(100);
    b.fly();
}
```

```
Aircraft ctor
Aircraft flying: 50
Aircraft ctor
Boeing ctor
Aircraft flying: 100
Boeing ~dtor
Aircraft ~dtor
Aircraft ~dtor
```

Construction / Destruction call order

- Construction
  - Base class constructor is called first then the constructor of the derived class
  - Whenever any constructor of a derived class (either default or with parameters) is called, the default constructor of the base class is called automatically and executed first
- Destruction
  - It works in exactly the opposite order of construction
  - Derived class destructor is called first then the destructor of the base class

Construction / Destruction order: example 1

```
class Aircraft
{
public:
    Aircraft() {cout << "Default Aircraft ctor" << endl;}

    Aircraft(int i)
    {
        capacity = i;
        cout << "Aircraft ctor with parameters" << endl;
    }

    ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}

    void fly() {cout<<"Aircraft flying: "<<capacity<< endl;}

protected:
    int capacity; //nbre of pass.
};
```

```
class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Default Boeing ctor" << endl;}

    Boeing(int i)
        {
            capacity = i;
            cout<<"Boeing ctor with parameters"<<endl;
        }

    ~Boeing(){cout << "Boeing ~dtor" << endl;}
};
```

```
// main function
int main()
{
    Boeing b(300);
    b.fly();
}
```

```
Default Aircraft ctor
Boeing ctor with parameters
Aircraft flying: 300
Boeing ~dtor
Aircraft ~dtor
```

Construction / Destruction order: example 2

```
class Aircraft
{
public:
    Aircraft() {cout << "Default Aircraft ctor" << endl;}

    Aircraft(int i)
    {
        capacity = i;
        cout << "Aircraft ctor with parameters" << endl;
    }

    ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}

    void fly() {cout<<"Aircraft flying: "<<capacity<< endl;}

protected:
    int capacity; //nbre of pass.
};
```

```
class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Default Boeing ctor" << endl;}

    Boeing(int i):Aircraft(i)
        {
            capacity = i;
            cout<<"Boeing ctor with parameters"<<endl;
        }

    ~Boeing(){cout << "Boeing ~dtor" << endl;}
};
```

```
// main function
int main()
{
    Boeing b(300);
    b.fly();
}
```

```
Aircraft ctor with parameters
Boeing ctor with parameters
Aircraft flying: 300
Boeing ~dtor
Aircraft ~dtor
```

Types of inheritance

- Derived classes can inherit a base class in three different fashions
  - Public
    - Derived class keeps the same access rights to the inherited members
    - Public members in base class remain public in derived class
    - Protected members in base class remain protected in derived class
  - Private
    - Derived class changes the accessibility rights to the inherited members
    - Public and protected members in base class become private in derived class
  - Protected
    - Derived class changes the accessibility rights to the inherited members
    - Public and protected members in base class become protected in derived class

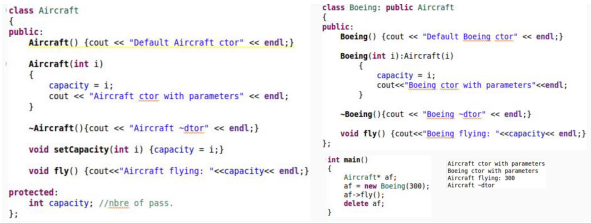
Architecture dilemma: is-a VS has-a

- When designing the classes of a software you should define carefully the relationship between those classes
  - Should class A inherit from class B or should it contain a pointer to class B?
  - Should class Aircraft inherit from class Engine since every aircraft has an engine?
- If A is B then A should inherit from B
- If A has B as one of its components then A should contain B and not inherit from it

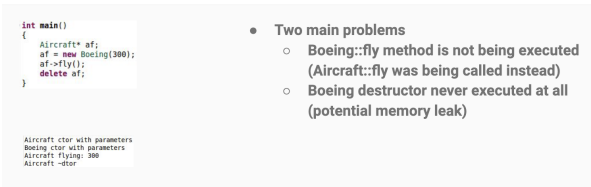
Few words about multiple inheritance

- C++ allows a class to inherit from multiple other classes
  - class FighterJet : public Aircraft, public Fighter
- Order of construction follows the same order of declaration
  - Aircraft ctor then Fighter ctor, then FighterJet ctor
- Beware the diamond problem
  - Use virtual inheritance to avoid the headache

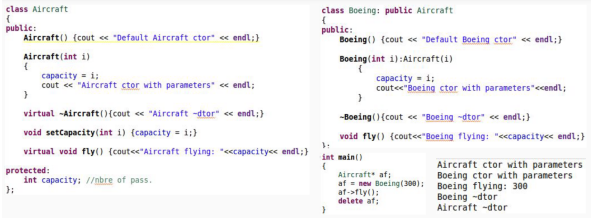
Polymorphism: having different forms



Polymorphism



Polymorphism: virtual methods



Polymorphism: virtual keyword

- Always mark destructor virtual if the class is meant to be inherited
- You only need to mark the destructor of the base class virtual. By doing so, the compiler will automatically consider all subclasses' destructors as virtual as well.
- You only need to mark the polymorphic methods in the base class as virtual. However, it is common to mark them virtual in the derived classes as well for readability.
- C++11 introduced the keyword "override" to enhance the readability of the polymorphic methods

Virtual methods VS pure virtual methods

- Virtual method has an implementation in the base class and can be overridden by a derived class to obtain polymorphic behavior
- Pure virtual method does not have an implementation in the base class and should necessarily be implemented in the derived classes
  - virtual void fly ()= 0;
- Class that does have at least one pure virtual method is called an abstract base class (similar to Java's interface classes)
- Abstract base classes cannot be instantiated. Only derived classes can

COMP 322 Lecture 9 - Exception Handling

Junji Duan

2024/3/15

Today's Outline

- What are exception
- Try . . . catch
- Layers of exceptions

What's an exception?

- Exception is an unexpected behavior
- Exceptions are not necessarily errors (bugs), they are more about forgetting to handle errors
- Called exceptions because they deal with exceptional circumstances that may arise during runtime
- Exceptions are raised when there is no logical way for a method to continue its execution
- Object oriented way of implementing "error codes" used in plain C language

example:

- What if malloc or new failed to provide the demanded memory block?
  - Remember that a system has a limited memory that may run out
- What if your code reads a file that is supposed to be present (and you as a programmer are taking for granted that it is always present). One sunny day, someone deleted that file ...
- What if your code reads a feed from a website, then one rainy day the site's server went down

Try . . . catch . . . throw

- Exceptions should be caught when they occur
  - Using a try and catch blocks
  - Portion of code to be monitored for exceptions should be enclosed within the try block (using try keyword)
  - Exception handling is done within the catch block (using the catch keyword)
  - To signal an exception or to propagate it to an outer code level, we use the throw keyword
  - Unlike Java, C++ does NOT support a "finally" block. Whatever code that "finally" must have, should be done in the destructor.

```
int main()
{
    try
    {
        // Portion of code to be
        // monitored for exceptions
    }
    catch(...)
    {
        // Exception handling is done here
    }
}
```

```
double getRatio(double a, double b)
{
    return a/b;
}

int main()
{
    double ratio1 = getRatio(5, 25);
    double ratio2 = getRatio(5, 0);
}
```

- getRatio should *throw* an exception if provided with zero value for *b*

```
double getRatio(double a, double b)
{
    if (b == 0)
    {
        throw "Warning: Division by Zero";
    }
    return a/b;
}

int main()
{
    try
    {
        double ratio1 = getRatio(5, 25);
        double ratio2 = getRatio(5, 0);
    }
    catch (const char* message)
    {
        cout << message << endl;
    }
}
```

- Catch argument type should match the throw argument type
- In our example, the type is **const char*** because this is exactly the type of the message that we sent using **throw**

```
double getRatio(double a, double b)
{
    if (b == 0)
    {
        string msg = "Warning: Division by Zero";
        throw msg;
    }
    return a/b;
}

int main()
{
    try
    {
        double ratio1 = getRatio(5, 25);
        double ratio2 = getRatio(5, 0);
    }
    catch (string& message)
    {
        cout << message << endl;
    }
}
```

- We can use string as well ...

C++ standard exceptions

- #include
- C++ standard library offers a list of exceptions
  - std::bad_alloc
  - std::out_of_range
  - ...
- C++ offers also a base class to create user defined "object" exceptions
  - Define new exceptions by inheriting from the base class **std::exception**

Inheriting from class exception: Example

```
class ZeroException: public exception
{
    virtual const char* what() const throw()
    {
        return "Warning: Division by Zero";
    }
};
ZeroException divideByZeroException;

double getRatio(double a, double b)
{
    if (b == 0)
    {
        throw divideByZeroException;
    }
    return a/b;
}

int main()
{
    try
    {
        double ratio1 = getRatio(5, 25);
        double ratio2 = getRatio(5, 0);
    }
    catch (exception& e)
    {
        cout << e.what() << endl;
    }
}
```

- std::exception has a virtual member method called **what()** that can be reimplemented in derived classes to personalize a user message about the cause of the exception
- **const throw()** part of the declaration:
  - **const**: the method will not alter the state of the class
  - **throw()**: for C++98 means that this method is guaranteed not to throw exceptions
    - Replaced by **noexcept** since C++11

Catching multiple exceptions

```
try
{
    // code that might through some exceptions
}
catch ( ZeroException& e )
{
    // handling division by zero exception
}
catch ( SomeOtherCostumException& e )
{
    // handling some other user defined exception
}
catch ( const std::exception& e )
{
    // handling all other standard exceptions
}
catch ( ... )
{
    // handling non defined unexpected exceptions
}
```

- We can manage different type of exceptions separately
- Respect the order of catching
  - Derived classes first, then base class
  - (...) at the end

Try catch blocks can be nested - 1

```
class ZeroException: public exception
{
public:
    virtual const char* what() const throw()
    {
        return "Warning: Division by Zero";
    }
};
ZeroException divideByZeroException;

class SomeOtherException: public exception
{
public:
    virtual const char* what() const throw()
    {
        return "Some Other Exception";
    }
};
SomeOtherException otherException;
```

```
double getRatio(double a, double b)
{
    if (b == 0)
    {
        throw divideByZeroException;
    }
    return a/b;
}
```

Try catch blocks can be nested - 2

```
int main()
{
    try
    {
        try
        {
            double ratio1 = getRatio(5, 25);
            double ratio2 = getRatio(5, 0);
        }
        catch (ZeroException& e)
        {
            cout << e.what() << endl;
            throw otherException;
        }
    }
    catch (exception& e)
    {
        cout << "Outer catch: " << e.what() << endl;
    }
}
```

```
Warning: Division by Zero
Outer catch: Some Other Exception
```

Catch me if you can ;) - take 1

```
int main()
{
    try
    {
        try
        {
            double ratio1 = getRatio(5, 25);
            double ratio2 = getRatio(5, 0);
        }
        catch (SomeOtherException& e)
        {
            cout << "Inner catch: " << e.what() << endl;
        }
    }
    catch (exception& e)
    {
        cout << "Outer catch: " << e.what() << endl;
    }
}
```

- Which catch will catch the ZeroException?
  - The outer catch. Exceptions propagate to the outer blocks.

Catch me if you can ;) - take 2

```
int main()
{
    try
    {
        try
        {
            double ratio1 = getRatio(5, 25);
        }
        catch (SomeOtherException& e)
        {
            cout << "Inner catch: " << e.what() << endl;
        }
    }
    catch (exception& e)
    {
        cout << "Outer catch: " << e.what() << endl;
    }
}
```

- Which catch will catch the ZeroException?
  - None of them ;)

Catch me if you can ;) - take 3

```
int main()
{
    try
    {
        try
        {
            double ratio1 = getRatio(5, 25);
        }
        catch (ZeroException& e)
        {
            cout << "Inner catch: " << e.what() << endl;
        }
    }
    catch (exception& e)
    {
        cout << "Outer catch: " << e.what() << endl;
        throw divideByZeroException;
    }
}
```

- Which catch will catch the ZeroException? (assuming that the outer catch caught an exception and it threw divideByZeroException)
  - None of them. Exceptions do not propagate to the inner blocks.

Catch me if you can ;) - take 4

```
int main()
{
    try
    {
        try
        {
            double ratio1 = getRatio(5, 25);
            double ratio2 = getRatio(5, 0);
        }
        catch (ZeroException& e)
        {
            cout << "Inner catch: " << e.what() << endl;
            throw divideByZeroException;
        }
    }
    catch (exception& e)
    {
        cout << "Outer catch: " << e.what() << endl;
    }
}
```

- Which catch will catch the ZeroException?
  - getRatio(5, 0) will trigger the inner catch.
  - throw divideByZeroException will trigger the outer catch

Exception handling and control transfer

- When a program throws an exception the execution control is transferred to the catch block and never returns to the block that threw the exception
- If an exception occurs and the program does not provide exception handlers or if it does provide one but the catch block exception declaration is not of the same type as the thrown object, the program will abort.
- When the control is transferred from a throw-point to a handler, destructors are invoked for all automatic objects constructed since the try block was entered

COMP 322 Lecture 10 - Templates

Junji Duan

2024/3/22

Today's Outline

- Generic programming
- C++ Templates
- Function templates
- Class templates
- STL quick overview

Example: same function, different data-types

```
int getMax(int a,  int b)
{
    return (a > b) ? a : b;
}

float getMax(float a,  float b)
{
    return (a > b) ? a : b;
}

double getMax(double a,  double b)
{
    return (a > b) ? a : b;
}
```

- Same code is duplicated in order to support different data types

What is generic programming?

- A programming paradigm in which types are abstracted and common code for all types is written only once
- Source code will be processed and transformed by the compiler before generating the object code
- Less work on the programmer's side, more work on the compiler's side
- C++ implements generic programming via the use of "Templates"
- Also called meta-programming (code that generates code)
- Java's Generics is the closest thing to C++ templates

Example: same function, different data-types

```
template <typename T>
T getMax(T a,  T b)
{
    return (a > b) ? a : b;
}

int main()
{
    float myFloatMax = getMax<float>(12.1, 120.3);
    int   myIntMax   = getMax<int>(15, 7);
}
```

- By using a template, the compiler will generate the needed code to support all needed types
- The type is abstracted as T
- Templates applied to functions are called function templates

C++ Templates

- both declaration and definition of a template must reside in the same file (either .h or .cpp)
- Compiler generates object code for generic source code on an "as-needed" basis
- `template <typename T>` is equivalent to `template <class T>`
  - You can use `typename` or `class` interchangeably

Templates can support multiple types

```
template <typename T, typename U>
void printValues(T a,  U b)
{
    cout << "first value: " << a ;
    cout << ", and second value: " << b;
    cout << endl;
}

int main()
{
    printValues<int, double>(9, 12.11);
}
```

- T and U may or may not have the same type
- Templates can support as many different typenames as needed

Templates can be provided a default type

```
template <typename T, typename U=double>
void printValues(T a,  U b)
{
    cout << "first value: " << a ;
    cout << ", and second value: " << b;
    cout << endl;
}

int main()
{
    printValues<int>(9, 12.11);
}
```

- U will be substituted by "double" if no type input was provided to the template

Templates can be specialized for specific types

```
template <typename T>
T subtract(T a,  T b)
{
    return a-b;
}

template<>
string subtract(string a, string b)
{
    size_t pos = a.find(b);
    string str = a.substr (0, pos);
    return str;
}
```

```
int main()
{
    cout << subtract<int>(9, 12.11) << endl;

    string a = "Hello world";
    string b = "world";
    cout << subtract<string>(a, b);
}
```

```
-3
Hello
```

- The subtract function template has different implementation in the case of string type

Class Templates

```
template <typename T>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = a;
        y = b;
        z = c;
    }
};

int main()
{
    Coordinates<int>    pointi(2, 4, -3);
    Coordinates<float>  pointf(2.1, 4.9, -3.7);
}
```

- Templates can be applied to classes

```
template <typename T>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = a;
        y = b;
        z = c;
    }

    T getX ()
    {
        return x;
    }
};

int main()
{
    Coordinates<int>    pointi(2, 4, -3);
    Coordinates<float>  pointf(2.1, 4.9, -3.7);
    cout << pointf.getX();
}
```

- Method members can be defined within the class

```
template <typename T>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = a;
        y = b;
        z = c;
    }

    T getX();
};
```

```
template <typename T>
T Coordinates<T>::getX ()
{
    return x;
}
```

- Method members can be defined outside of the class, but be careful with the syntax

Class Templates: non-type parameters

```
template <typename T, int scale=1>
class Coordinates
{
    T x, y, z;
public:
    Coordinates(T a, T b, T c)
    {
        x = scale*a;
        y = scale*b;
        z = scale*c;
    }

    T getX();
};

template <typename T,  int scale>
T Coordinates<T, scale>::getX ()
{
    return x;
}
```

```
int main()
{
    Coordinates<int>    pointi(2, 4, -3);
    Coordinates<float, 2>   pointf(2.1, 4.9, -3.7);
    cout << pointf.getX();
}
```

- Templates can also accept regular parameters (similar to the way functions behave)

Templates: pros

- Avoid code duplication
  - which reduces development time
  - And also provides more safety when updating and maintaining the code
- Preferable to C-macros because they provide type safety
- Preferable to deriving classes when performance is on stake
  - In other terms, compile time polymorphism is prefered to run time polymorphism when efficiency is terribly needed

Templates: cons

- Heavy use of templates may result in "code bloating" which is the generation of huge object files
  - Which also means long compilation and build time
  - Which also means a huge executable files
- You lose code hiding when using templates because you have to provide the implementation in the header file
- Templates had the reputation of being hard to debug due to cryptical error messages provided by compilers
  - Since the compiler generates additional code for templates, it is hard to locate during run time the origin of an error when debugging
  - However, compilers had made a lot of enhancements since ...

Templates: pros

- This is exactly how you would look like when debugging code with heavy class template usage

The Standard Template Library (STL)

- STL's design and architecture is credited largely to Alexander Stepanov

  - Alex is a computer scientist and generic programming advocate

- STL was not part of the original standard library (it was added later on)
- STL provides a set of class templates implementing the basic data structures
- STL main components are: containers, iterators, algorithms and functions

STL components: Containers

- Generic classes to store objects and data
- Sequential containers:
  - list
  - Vector
  - arrays (since C++11)
  - ...
- Associative containers:
  - map
  - set
  - ...

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

int main()
{
    list<int> intList;
    intList.push_front(12);
}
```

STL components: Iterators

```
int main()
{
    list<int> intList;
    list<int>::iterator it;

    intList.push_front(12);
    intList.push_front(22);
    intList.push_front(46);

    for (it=intList.begin(); it!=intList.end(); ++it)
    {
        cout << *it << endl;
    }
}
```

- Iterators make it possible to iterate over containers and accessing their values
- Iterators are abstraction to access different types of containers

STL components: Algorithms

- STL provides a large collection of algorithms to be applied on containers
  - Sorting
  - Searching
  - Comparing
  - ...

STL components: Functions

- STL offers classes that overload the function call operator: operator()
- This is a C++ techniques that is also known under the name of: Functors
  - Functor = function object
  - The idea is that regular functions don't keep state. Objects on the other hand do, so by overloading the function call operator() we'll be imitating the function behavior while keeping the object's state