Lecture 3

TOC

1. 预处理命令和宏
2. 名称空间(namespace)
3. 作用域(scope)
4. 动态内存管理
5. 生命周期(lifetime)
6. 左值，右值
7. 别名(Alias)
8. auto 自动类型推导
9. Assignment1

预处理命令和宏

以 `#` 开头，每句占一行或以 `\` 拼接下一行。

```
#pragma once

#include <iostream>

#if defined(__cplusplus)
extern "C" {
# elif defined(_WIN32) || defined(_WIN64)
// some code
# endif

#ifdef __clang__
// some code
#endif

#define sum(a, b) \
do {              \
    (a) + (b);    \
} while(0)

#define ONE 1 /*useless*/
#undef sum
```

注释

- 行注释：`//`
- 块注释：`/* */`

```
int foo = 1; // this is a line comment
int a = /* this is a block comment */ 2;
```

函数

函数是一段封装了一系列操作的代码块，可以通过函数名调用。使用函数可以提高代码的复用性和可读性。

- 函数声明:

```
extern void bar();
int add(int, int);
int     foo   (int a, int b);
//|返回值|函数名|参数列表    |
//参数列表为空时，使用 `void` 表示，但一般省略。
//声明时可以省略参数名。
```

- 函数定义: 遵循单定义原则

```
int foo(){
 return 1;
```

C++ 函数重载

C++ 支持函数重载，允许在同一作用域中定义多个同名函数，但这些函数的参数列表必须不同。编译器根据函数调用时传递的参数个数和类型来选择合适的重载函数。

```
void foo(){
    std::cout<<"foo1\n";
}

void foo(int a){
    std::cout<<a<<'\n';
}

int main(){
    foo();
    foo(1);
}
```

- C capacity: C++ 函数重载会在编译期生成与原本不同的函数名，因此在与 C 混编时需要使用 `extern "C"` 修饰函数声明。

函数指针

函数指针是指向函数的指针变量，可以通过函数指针调用函数。

```
int add(int a, int b){
    return a+b;
}

int(*p)(int, int) = add;
int(*q)(int, int) = &add;

// invoke function
int res = p(1, 2);
int res2 = q(1, 2);
```

main 函数

main 函数是编译器默认的入口函数，是一个返回值为 `int` 类型，参数列表为空或 `int, char**` 的函数

```cpp
#include <iostream>

int main(int argc, char **argv) {
  for (int i = 0; i < argc; ++i) {
    std::cout << argv[i] << '\n';
  }
  return 0;
}
```

打印命令行参数

```
❯ g++ foo.cpp
❯ ./a.out hello world !
./a.out
hello
world
!
```

名称空间(namespace)

名称空间是 C++ 中用来避免命名冲突的一种机制，可以将一组标识符封装在一个作用域中。

```
#include <iostream>
namespace foo {
void bar() { std::cout << "namespace foo\n"; }
} // namespace foo

namespace fumo {
void bar() { std::cout << "namespace fumo\n"; }
struct A {
  int a;
};
} // namespace fumo

int main() {
  foo::bar();
  fumo::bar();
  fumo::A a;

  {
    using foo::bar;
    bar();
  }
  {
```

Internal Linkage

通过 `static` 关键字或使用**匿名名称空间**，我们可以指定变量或函数为内部链接，即仅本编译单元可见(一般为当前的 .cpp 文件)。

File "a.cpp"

```
#include <iostream>
static void foo() { std::cout << "this is foo in a\n"; }
```

File "b.cpp"

```
#include <iostream>

void foo() { std::cout << "this is foo in b\n"; }
int main() {
  foo();
  return 0;
}
```

作用域(scope)

C++ 中，作用域有如下几种:

- Global scope
- Block scope
- Function parameter scope
- more...

```cpp
#include <iostream>

int global_var = 1;

int main(int argc, char** argv) {
  {
    int global_var = 2;
    std::cout << global_var << '\n';
  }
  std::cout << global_var << '\n';
}
```

Reference

动态内存管理

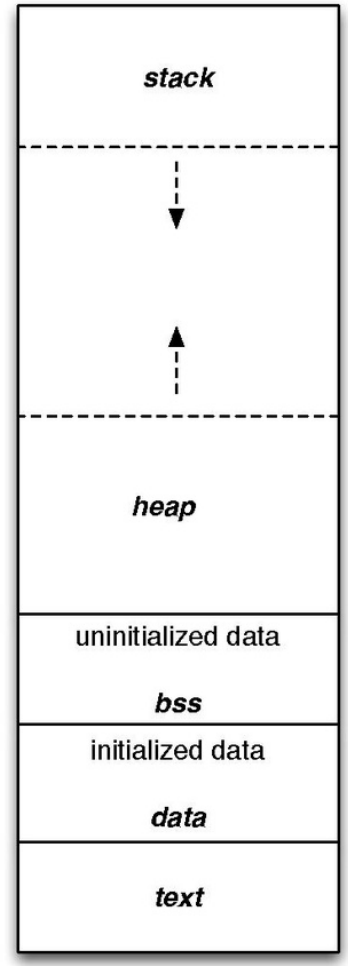
动态内存分配是指在程序运行时根据需要分配和释放内存，而不是在编译时确定内存的大小。程序运行时可以从堆上动态申请内存。

- C：`malloc` and `free`
- C++: `new` and `delete`

```
char* name=malloc(sizeof(char)*100);
assert(NULL≠name);

int* id = new int;
int* class_id = new int[10];

free(name);
delete id;
delete class_id;
```



生命周期(lifetime)

C++ 中变量的生命周期大约有以下几种：

- Automatic storage duration
- Static storage duration
- Thread storage duration
- Dynamic storage duration

```
int global_var = 1; // static storage duration
static int static_var = 2; // static storage duration

int main() {
  int local_var = 3; // automatic storage duration
  static int static_local_var = 4; // static storage duration
  std::thread t([]() {
    int thread_local_var = 5; // thread storage duration
  });
  int* dynamic_var = new int(6); // dynamic storage duration
}
```

存储周期关键字

- auto: c++11前表示自动生命周期，c++11后表示自动类型推导
- register：表示变量应该存储在寄存器中，已经被废弃
- extern：表示变量具有外部链接性
- static：表示变量具有内部链接性或持久性

Reference

左值，右值

C++11 引入了右值引用的概念，使得我们可以更好地理解对象的生命周期。

- 广义左值（glvalue）：可以取地址的表达式，在赋值运算中位于等号左侧
  - 左值(lvalue): 可以取地址、持久存在的对象。
  - 亡值(xvalue): 即将销毁，资源可以被重用的对象。
- 右值：不能持有持久存储位置的表达式，它们通常是临时对象或常量，不能出现在赋值操作的左边。右值是表达式的结果或者是常量值。
  - 纯右值(prvalue)：无法取地址的临时对象或常量，比如字面量、临时对象等。
  - 亡值(xvalue)

Reference

移动语义 `std::move()`

`std::move()` 返回指向当前左值的右值引用，用于将左值转换为右值，从而实现资源的转移。这在类的构造中非常有用，能减少资源拷贝的开销。

```
int a = 10;
int&& r_ref = std::move(a);
```

引用坍缩和完美转发

- 模板元编程中常用，这里不过多展开。感兴趣可以自行学习。

Link

别名(Alias)

C 语言中，使用 `typedef` 关键字定义别名。C++11 引入了 `using` 关键字，提供了更多对模板的支持。

```cpp
#include <cstdint>

typedef int32_t i32;
using u8 = uint8_t;

typedef void (*Func)(int);

template <typename T1, typename T2> struct foo {
  T1 a;
  T2 b;
};

typedef foo<int, double> TypeDefFoo;

template <typename T1> using Using = foo<T1, int>;
```

auto 自动类型推导

C++11 改变了 auto 关键字的用法，使得 C++ 能够像其他新兴语言一样进行类型推导。

```
auto a = 1; // int
auto b = 1.0; // double
auto c = "hello"; // const char*
```

部分应用：

- Range-based for

```
int arr[] = {1, 2, 3, 4, 5};
for (auto i : arr) {
  std::cout << i << '\n';
}
```

- 新的函数定义写法

```
auto foo() → int {
  return 1;
}
```

- Universal Reference

使用 `auto&&` 可以实现完美转发，延迟类型推导，避免不必要的拷贝。

```
auto&& res = foo();
```

Assignment1

完成一个n维向量库:

- https://github.com/ARTINX/2025-Vision-Train-Lecture/tree/main/assignment/Assignment1

提交方式：

- 发送到邮件到 submit@vollate.top，标题为 `ARTINX2025视觉Assignment1`
- 代码以附件提交，需要压缩为一个 zip 文件，附件名为 `姓名-学号-Assignment1.zip`，自行替换为自己的姓名学号

> 务必按照以上格式提交，否则可能正常接收

拓展阅读

- Compile Phase
- 编译之法

Q&A