Lecture 3

TOC

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
```

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;
}
```

- 内联函数和 `inline` 关键字：在C/C++早期，inline关键字用于指定函数为内联函数。现代编译器会自动决定是否内联。 `inline` 关键字允许定义出现在多个编译单元中而不会产生链接错误。Reference

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

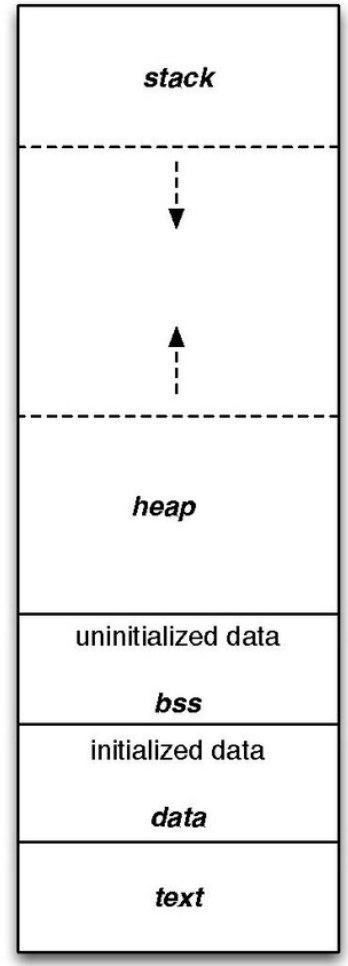
动态内存分配是指在程序运行时根据需要分配和释放内存，而不是在编译时确定内存的大小。程序运行时可以从堆上动态申请内存。

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

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

int* id = new;
int* 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

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`

拓展阅读

- Compile Phase

Q&A