C++ 笔记

shiyu-hong

2025-04-06

目录

欢迎

欢迎阅读这份 C++ 学习笔记!

第一章　未定义行为

在 C++ 中，**未定义行为（Undefined Behavior，UB）**特指违反语言规范的代码操作，其具体表现未在 C++ 标准中明确定义。此类行为虽然能够通过编译，但可能引发程序崩溃、产生错误输出，甚至因编译器实现差异或硬件特性导致完全不可预知的运行结果。

1.1　未初始化变量

在 C++ 中，**未初始化变量**是指声明变量后未显示赋初值，直接访问其内存残留数据的操作。

1.1.1　未初始化局部变量（最常见风险之一）

```cpp
#include <iostream>

int main(int argc, char **argv) {
  int x;        // 未初始化局部变量。
  int y{10};    // 初始化局部变量。
  int z{x + y}; // 危险：变量 x 未初始化，可能包含随机垃圾值。

  // 危险：输出随机垃圾值。
  // MSVC Debug: -840574134
  // MSVC Release: 10
  std::cout << z << std::endl;

  return 0;
}
```

1.1.2　未初始化条件变量

```cpp
#include <iostream>

void check_condition() {
  bool flag; // 未初始化条件变量。
```

```
  // 危险：条件判断可能随机成立。
  // MSVC Debug: Flag is true!
  // MSVC Release: Flag is false!
  if (flag) {
    std::cout << "Flag is true!" << std::endl;
  } else {
    std::cout << "Flag is false!" << std::endl;
  }
}

int main(int argc, char **argv) {
  check_condition();

  return 0;
}
```

1.1.3 未初始化数组

```
#include <iostream>

void process_array() {
  int buffers[3]; // 未初始化数组

  // 危险：操作未初始化的值。
  for (auto i = 0; i < 3; ++i) {
    buffers[i] += 1;
  }

  // 危险：输出随机垃圾值
  // MSVC Debug: 128545369 32760 128545369
  // MSVC Release: 8 1 1
  for (auto i = 0; i < 3; ++i) {
    std::cout << buffers[i] << " ";
  }
}

int main(int argc, char **argv) {
  process_array();

  return 0;
}
```

1.1.4 未初始化类成员变量

```cpp
#include <iostream>

class Point {
public:
  void print() { std::cout << "(" << x_ << ", " << y_ << ")" << std::endl; }

private:
  int x_; // 未在构造函数中初始化。
  int y_; // 未在构造函数中初始化。
};

void log_point() {
  Point point; // 未显示初始化成员。

  // 危险：输出随机垃圾值。
  // MSVC Debug: (-1289063848, 32759)
  // MSVC Release: (0, 0)
  point.print();
}

int main(int argc, char **argv) {
  log_point();

  return 0;
}
```

1.1.5 未初始化指针（最常见风险之一）

```cpp
#include <iostream>

int main(int argc, char **argv) {
  int *ptr; // 未初始化指针。
  *ptr = 3; // 危险：可能覆盖随机内存，触发段错误。

  return 0;
}
```

1.2 内存未定义行为

在 C++ 中，**内存未定义行为**指程序通过非法方式操作内存资源，导致 C++ 标准无法为其执行结果提供任何保证的行为。此类行为直接违反内存安全规则，可能引发程序崩溃、数据损坏或安全漏洞，且其

具体表现高度依赖编译器实现、操作系统及硬件环境。

1.2.1 解引用空指针

在 C++ 中，**空指针（Null Pointer）**是一个特殊指针值，表示指针不指向任何有效的对象或内存地址。它确保指针处于“未指向任何内容”状态的明确标识。若对其进行解引用（访问或修改内存），会导致未定义行为，可能引发程序崩溃、数据损坏或难以调试的逻辑错误。

```cpp
#include <iostream>

int main(int argc, char **argv) {
  // 空指针。
  int *ptr = nullptr;
  // 危险：将数据写入空指针地址。
  *ptr = 3;
  // 危险：读取空指针内容。
  std::cout << *ptr << std::endl;

  return 0;
}
```

1.2.2 解引用野指针

在 C++ 中，**野指针（Dangling Pointer）**是指向已释放或无效内存地址的指针。这些指针不再合法，但依然保留原来的地址值。若对其进行解引用（访问或修改内存），会导致未定义行为，可能引发程序崩溃、数据损坏或难以调试的逻辑错误。

```cpp
#include <iostream>

// （1）释放指针后未将其置空会形成野指针。
void foo1() {
  // 动态分配内存。
  int *ptr = new int(10);
  // 内存释放后，指针 ptr 仍保留原内存地址，成为野指针。
  delete ptr;
  // 危险：访问野指针指向的内存可能引发段错误，导致程序异常终止。
  *ptr = 3;
  // 危险：访问无效内存区域可能导致读取到垃圾值或引发段错误。
  std::cout << *ptr;
}

// （2）函数返回局部变量的内存地址会导致未定义行为。
int *create_dangling_pointer() {
  // 局部变量在栈上分配。
  int x{5};
```

```
  // 危险：函数返回后，其栈帧中的局部变量会被自动销毁，
  // 此时返回的指针将指向无效的栈内存，访问该指针会导致未定义行为。
  return &x;
}

void foo2() {
  int *ptr = create_dangling_pointer();
  // 危险：访问已释放的栈内存可能导致读取到垃圾值或引发段错误。
  std::cout << *ptr;
}

// （3）多个指针指向同一内存。
void foo3() {
  // 动态分配内存。
  int *p1 = new int(3);
  // p1 和 p2 指向同一块内存。
  int *p2 = p1;
  // 内存释放后，指针 p1 和 p2 仍保留原内存地址，成为野指针。
  delete p1;
  // 危险：访问野指针指向的内存可能引发段错误，导致程序异常终止。
  *p2 = 3;
}

// （4）数组越界访问。
void foo4() {
  int arr[3]{1, 2, 3};
  int *ptr = &arr[0];
  // 危险：指针越界访问可能指向未分配的未知内存区域，导致未定义行为或内存访问冲突。
  ptr += 5;
  // 危险：访问野指针指向的内存可能引发段错误，导致程序异常终止。
  *ptr = 4;
}

// （5）对象成员指针失效。
struct Foo {
public:
  int *data;

public:
  Foo() { data = new int(3); }

  ~Foo() { delete data; }
};
```

```
void foo5() {
  Foo foo;
  // 复制指向数据的指针。
  int *ptr = foo.data;
  // 显示释放内存后未置空指针（或在对象析构后未处理成员指针），导致产生野指针。
  delete foo.data;
  // 危险：访问野指针指向的内存可能引发段错误，导致程序异常终止。
  *ptr = 1;
}

int main(int argc, char **argv) {
  foo5();
  return 0;
}
```