



ОНЛАЙН-ОБРАЗОВАНИЕ


Онлайн-образование

Проверить, идет ли запись!





Меня хорошо видно && слышно?

Ставьте  , если все хорошо
Напишите в чат, если есть проблемы

Обработка ошибок в С



Легкоступ Виктор Валерьевич

Преподаватель



Легкоступ Виктор

- Специализация: фильтрация данных, оценивание параметров систем, системы автоматического управления, обработка сигналов, численные методы, аэродинамика, параллельные вычисления.
- Базовые инструменты: C/C++, Python, Matlab/Simulink, Mathematica
- Профессиональные интересы: БЛА, системы управления и измерения, моделирование

The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue and green gradient. A network of thin, light blue lines connects various points across the gradient, creating a digital or technological feel. The word "Начало" is centered in the middle of the image in a white, sans-serif font.

Начало

План

План:

1. Зачем это надо
2. Виды ошибок
3. Исключительные ситуации и ожидаемое поведения
4. Способы обработки ошибок и исключений C++ (упомянуть про SEH)
5. Setjmp/longjmp и стек
6. Утверждения assert и static_assert
7. Логирование в файл и консоль
8. Необрабатываемые ошибки и завершение программы
9. Точки останова и отладка

Зачем это надо?

Чтобы создавать надежный и устойчивый к отказам код, нам необходимо иметь представление о таких вещах, как:

1. Окружающая среда выполнения (аппаратно-программного комплекса);
2. Места кода, потенциально приводящих к ожидаемой/непредвиденной ситуации, способной привести к отказу нормального функционирования нашего программного продукта;
3. Способы обработки таких ситуаций.

На этом занятии мы рассматриваем лишь не выходящие за пределы языка C техники.

Виды ошибок

1. Ошибки времени компиляции
2. Ошибки времени выполнения
3. Логические ошибки

Ошибки, происходящие во время работы программы можно классифицировать:

1. Ошибка определения данных
2. Ошибка проектирования
3. Ошибка кодирования
4. Ошибка накопления погрешности

Способы обработки возможных ошибок

1. Проверка на корректность значений по месту их появления
2. Выделение кода в функцию, которая в случае ошибки возвращает флаг наличия ошибки
3. Выделение кода в функцию, которая в случае ошибки возвращает код ошибки
4. Выделение кода в функцию, которая в случае ошибки возвращает объект
5. Выделение кода в функцию, которая в случае ошибки устанавливает глобальную переменную
6. Выделение кода в функцию, в которую также передается callback-функция-обработчик потенциальных ошибок
7. Использование обработки исключений try/catch (C++)
8. Выход из функции с помощью setjmp/longjmp
9. Использование архитектурных средств (Windows/Linux/OS X)

Что следует внимательно проверять

1. Результат выделения памяти, ресурсов
2. Результат открытия ресурса (файл, сокет, память, устройство, консоль,...)
3. Входные данные (от пользователя, из файла, из сети, ...)
4. Делитель на околонулевое значение (нулевое для целых чисел)
5. Результат работы сложного алгоритма (в разумных пределах)
6. Результат выполнения какой-либо операции (системной/библиотечной)

Проверка значений на корректность в runtime

Все, что вводит человек – это потенциально проблемные данные, которые необходимо проверить. То же касается различных файлов, устройств ввода и т.д., управляемых или формируемых человеком.

```
printf("Желаете продолжить [y/n]: ");  
c = getche();  
if (c == 'y')  
    ...  
else if (c == 'n')  
    ...  
else  
    printf("Недопустимый символ. Повторите ввод: ");
```

Проверка значений на корректность при отладке

Часто при отладке приходится много переменных проверять на соответствие ожидаемым величинам. При этом оператор `if ()` используется как для реализации логики программы, так и для сервисных рутин, типа проверки ошибок и т.д. Такие рутины можно реализовать через `assert ()`, что сделает проверку значений более явной и заметной, а в будущем ее легко можно будет отключить.

```
#include <assert.h>

...
assert(inflation_rate > 1000 && "Error. You should escape");
...
```


Проверка значений на корректность

Также в C11 есть `static_assert()` и `_Static_assert()` для проверки значений на этапе компиляции.

```
#include <assert.h>

...
static_assert(eyes_number > 2, "Error. You are not a human");
...
```

Возврат флага ошибки из функции

Довольно эффективный способ определить сам факт наличия ошибки. Из минусов – после вызова функции требуется обрабатывать ошибки, что загромождает логику алгоритма. Также нельзя вернуть из функции результат ей работы.

```
//--- файл func.c ---
bool func() {
    ...
    return true; // все хорошо
    ...
    return false; // ошибка вычислений
}

//--- файл main.c ---
int main() {
    ...
    if (!func()) {
        ... // тут обрабатываем ошибку
    }
}
```


Возврат кода ошибки из функции

Довольно эффективный способ определить как сам факт наличия ошибки, так и конкретную причину. Из минусов – после вызова функции требуется обрабатывать ошибки, что загромождает логику алгоритма. Также нельзя вернуть из функции результат её работы.

```
//--- файл func.c ---
int func() {
    ...
    return 0; // все хорошо
    ...
    return -1; // ошибка вычислений
    ...
    return -2; // нехватка памяти
}

//--- файл main.c ---
int main() {
    ...
    if (func()) {
        ... // тут обрабатываем ошибку
    }
}
```

Установка глобальной переменной

Весьма эффективный способ, так как позволяет возвращать из функции значения и при том обработку ошибок можно частично отделить от основной логики алгоритма.

Минус: легко забыть о проверке ошибок

```
static int last_error;

int func() {
    ...
    last_error = 3; // ошибка (нехватка памяти)
    ...
}

int main() {
    ...
    res = func();
    ...
    if (last_error) {
        ... // тут делаем проверку всевозможных ошибок, если не забыли
    }
}
```


Идем дальше

Вопрос: что тут не так?

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

int main() {
    ...
    z1 = div(x, y1);
    z2 = div(x, y2);
    z3 = div(x, y3);
    z4 = div(x, y4);
    ...
}
```

Обработка исключений (C++)

Данная конструкция используется в случае возможности существования исключительной ситуации, в результате которой программа не может продолжить нормально функционировать.

```
double div(double a, double b)
{
    if (fabs(b) < 0.00000001)
        throw "Dividing by zero in div()";
    return a/b;
}

int main() {
    ...
    try {
        z1 = div(x, y1);
        z2 = div(x, y2);
    }
    catch (const char* exception) {
        printf(exception);
        // освобождаем ресурсы
    }
    ...
}
```


Setjmp/longjmp

В языке С есть некое подобие обработке исключений – это setjmp/longjmp из библиотеки <setjmp.h>

```
<setjmp.h>

#define DIV_BY_ZERO 5
double div(double a, double b)
{
    if (fabs(b) < 0.00000001)
        longjmp(env_buf, DIV_BY_ZERO);
    return a/b;
}

jmp_buf env_buf;

int main() {
    ...
    if (DIV_BY_ZERO == setjmp(env_buf)) {
        printf("Divide by zero. Examine input values.");
        // освобождаем ресурсы
        exit(-1);
    }
    z1 = div(x,y1);
    z2 = div(x,y2);
    ...
}
```

Способы снизить количество ошибок

1. Включение всех предупреждений, их осмысление и устранение (по возможности).
2. Использование единого последовательного стиля программирования. Используйте осмысленные, самодокументирующие, короткие имена, которые позволяют: различать имена типов (классов), функций, объектов (возможно констант и указателей), отличать команды препроцессора. Следите за отступами. Используйте эффективные правила оформления операторов. Не бойтесь упорядочивать код с помощью пробелов как горизонтально, так и вертикально.
3. Ограничение областей видимости.
4. Минимальное использование глобальных переменных.
5. Хорошее комментирование кода (алгоритмы, переменные, типы, функции...).
6. Всегда инициализировать чем-то переменные (создание с инициализацией).
7. Перегрузка операторов только в случае веских причин.
8. Внимательная работа с кучей и массивами.
9. Исключения – эффективный инструмент (C++).
10. Отказ по возможности от множественного наследования (C++).
11. Прогон программы хотя бы раз под отладчиком пошагово.
12. Использование системы контроля версий.
13. Использование тестов (с умом).
14. Создавайте в файлах стандартные заголовки со всей важной информацией (назначение, и т.д.)
15. Профилирование

Задание – обработать все потенциальные ошибки

```
/* Данная программа генерирует заданное пользователем количество
случайных чисел, записывает их в файл и выводит на консоль их
арифметическое среднее. В процессе устранения недостатков допустимо
менять интерфейсы.
*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <setjmp.h>
```

```
int div_int(int a, int b)
{
    return a / b;
}
```

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

```
int calc_mean(const int * buf, int len)
{
    int mean = 0;
    for (int i = 0; i < len; i++)
    {
        mean += buf[i];
    }
    mean = div_int(mean, len);
    return mean;
}
```

```
void rand_save(const int* buf, int length, const char fname[])
{
    FILE* f = fopen(fname, "w");
    for (int i = 0; i < length; i++)
    {
        fprintf(f, "%1d, ", buf[i]);
    }
}
```

```
void rand_gen(int** buf, int length)
{
    *buf = (int*)malloc(length * sizeof(int));
    for (int i = 0; i < length; i++)
    {
        (*buf)[i] = rand() % 10;
    }

    rand_save(*buf, length, "D:\\rand.txt");
}
```

```
int main()
{
    int length;
    printf("Enter the length: ");
    scanf("%d", &length);
    //while (getchar() != '\n');

    int* buf;
    rand_gen(&buf, length);

    int mean = calc_mean(buf, length);
    printf("Mean = %d \n", mean);
}
```

Рефлексия

Вопросы?



Список литературы

Роббинс Д. - Отладка приложений для Microsoft .NET и Microsoft Windows

Столяров – Азы программирования, Парадигмы

Роберт Мартин - Чистый код

David Svoboda - Beyond errno Error Handling in C


https://www.tutorialspoint.com/cprogramming/c_error_handling.htm

<https://man7.org/linux/man-pages/man3/errno.3.html>


<https://stackoverflow.com/questions/385975/error-handling-in-c-code>

<https://stackoverflow.com/questions/14685406/practical-usage-of-setjmp-and-longjmp-in-c#14685524>

<https://softwareengineering.stackexchange.com/questions/64926/should-a-method-validate-its-parameters/65031#65031>

The background of the image is an aerial photograph of a dense city skyline, likely New York City, with numerous skyscrapers. The image is overlaid with a semi-transparent blue layer that features a white geometric network pattern of interconnected dots and lines. The text is centered within this blue layer.

Заполните, пожалуйста,
опрос о занятии по ссылке в чате

The background of the entire slide is an aerial photograph of a dense city skyline, likely New York City, with numerous skyscrapers. A semi-transparent blue overlay covers the image, featuring a subtle network pattern of white lines connecting dots, resembling a digital or social network. The text is centered in the middle of the slide.

Спасибо за внимание!
Приходите на следующие вебинары

Легкоступ Виктор Валерьевич