МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»

НГТУ НЭТИ

Кафедра прикладной математики

Расчетно-графическое задание

по дисциплине «Языки программирования и методы трансляции»

ФПМИ

Группа	ПМ-92
Вариант	10
Студенты	Кутузов Иван
	Иванов Владислав
Преподаватель	Еланцева И. Л.
Дата	10.06.2022

Новосибирск

Задание:

Подмножество языка C++ включает:

- данные типа int, float, массивы из элементов указанных типов;
- инструкции описания переменных;
- операторы присваивания в любой последовательности;
- операции +, – , *, = =, != , <, > .

Методы диагностики и исправления ошибок. ошибки выполнения: нахождение индекса массива вне области действия, целочисленное переполнение, попытка чтения за пределами файла.

Теория:

Во время прогона в программах могут возникать ошибки, которые нельзя предусмотреть в процессе компиляции. При прогоне могут возникать следующие типы ошибок:

- нахождение индекса массива вне области действия;
- целочисленное переполнение (вызванное, например, попыткой сложить два наибольших целых числа, допускаемых реализацией);
- попытка чтения за пределами файла.

В языках с динамическими типами до времени прогона нельзя обнаружить более широкий класс ошибок (ошибки употребления типов, ошибки, связанные с присвоением и другие).

Обычно компиляторы стараются предотвратить возможность возникновения таких ошибок до прогона. Одно из решений – дать исчерпывающую формулировку задачи, например результат деления на ноль определить как ноль, выходящий за пределы области действия, индекс считать эквивалентным какому-нибудь значению в пределах области действия, при попытке чтения за пределами файла выполнять некоторое стандартное действие и т.д.

Однако такая исчерпывающая формулировка задачи имеет свои «подводные камни»: могут остаться незамеченными ошибки программирования или ошибки данных. Программисты обычно не ожидают, что во время прогона их программ произойдет деление на ноль, - им об этом нужно сообщить. Тем не менее, нежелательно, чтобы из-за этого прерывалось выполнение программы. Компромиссное решение – напечатать сообщение об ошибке времени прогона, когда она возникает, но позволить программе выполнить какие-либо стандартные действия, чтобы она могла продолжать работу и находить дальнейшие ошибки.

В случае ошибки, возникающей в процессе прогона, не всегда можно четко объяснить программисту, что именно неправильно. К этому моменту программа уже транслирована в машинный код, а программисту понятны только ссылки на исходный текст. Поэтому система, работающая при прогоне, должна иметь доступ к таблице идентификаторов и другим таблицам и следить за номерами строк в исходной программе. Таблицы, требуемые для диагностики, к началу прогона могут уже не находиться в основной памяти, но в случае ошибки должны туда загружаться и

фиксировать профиль программы на это время. Эта информация позволяет локализовать место возникновения ошибки или, по крайней мере, блок (рамку), внутри которой возникла аварийная ситуация.

Целочисленное переполнение относится к явлению, которое происходит в определенных типах компьютерных данных, когда их знаки переключаются с положительного на отрицательный или наоборот, когда они достигают конца своих применимых диапазонов. В компьютерных диапазонах целочисленные типы данных имеют круговые диапазоны, и когда они достигают одного конца своего диапазона, они сразу же переходят на другой конец своего диапазона.

Целое число со знаком может содержать диапазон значений от -2^31 до (2^31)^-1. Это целое число не может иметь значение (-2^31)^-1; скорее, следующее число, к которому оно увеличивается, находится на другом конце диапазона: (2^31)-1. Изменение с отрицательного на положительное в конце его диапазона является примером целочисленного переполнения. Кроме того, целое число не может иметь значение 2^31 ; вместо этого это значение переключится на другой конец диапазона и станет -2^31 .
Это переполнение имеет значительные последствия при программировании. Массив может содержать только столько индексов, сколько позволяет целочисленный тип, а отрицательные индексы не учитываются. Если программист пытается создать массив, размер которого больше, чем позволяет целочисленный тип, могут возникнуть значительные ошибки памяти, поскольку переполнение целочисленного значения приведет к отрицательному индексу. Это особенно опасно в языках, где нет явной проверки границ для массивов, таких как C ++.

Когда происходит целочисленное переполнение, могут возникнуть связанные типы переполнения, такие как переполнение буфера, переполнение кучи и переполнение буфера стека. Во всех этих случаях целочисленное переполнение приводит к переполнению структур памяти большим количеством данных, чем эти структуры могут удержать. Эти переполнения в простых программах нередко делают гораздо больше, чем вызывают неправильное чтение или неправильную запись. Однако манипулирование этой проблемой хакерами может привести к ошибкам памяти, которые могут вызвать более серьезные проблемы.

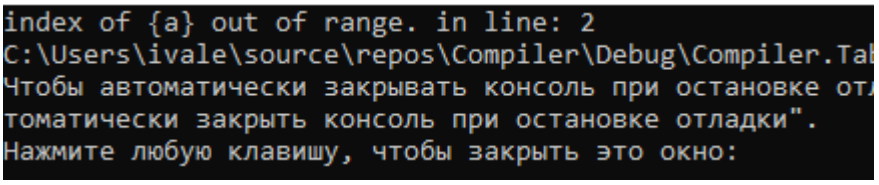
В большинстве простых программ целочисленное переполнение не является большой проблемой. Пределы целочисленного типа достаточно велики, чтобы проблема переполнения не возникала, если много данных не обрабатывается одновременно. В некоторых случаях переполнение может быть уменьшено, как в случае увеличения счетчиков, используя больший тип данных с большим диапазоном. Теоретически, больший тип данных может в конечном итоге столкнуться с той же проблемой переполнения, но с увеличением диапазонов типов данных шансы на это уменьшатся. Диапазон каждого целочисленного типа данных по крайней мере вдвое больше размера следующего наименьшего, поэтому имеется достаточно места для дополнительных данных.

Тест:

Индекс за пределами допустимого

```
void main() {
    int[] a = {1, 2, 3, 4, 5};
    a[5] + 4;
}
```

Ошибка:



Текст программы:

Syntax.h

```
case State::CAST_ARRAY_DEREFERENCE: {
    auto rule = _table->rule();

    auto index = _commandStack.getTokensByRule(rule).front();

    auto numberOfElements = arrays.get(arrayOnState.key);

    if (std::stoi(index.key) < 0 || std::stoi(index.key) >= numberOfElements) {
        errorMessage += "index of {" + arrayOnState.key + "} out of range. in line:
" + std::to_string(numberOfCommand);
    }

    result.push_back(index);

    break;
}

case State::CAST_ARRAY_IDENTIFER: {
    auto rule = _table->rule();

    auto identifer = _commandStack.getTokensByRule(rule).front();

    if (identiferTable.contains(identifer.key)) {
        errorMessage += "syntax error: re-declaring a variable {" + identifer.key +
"} " + "in line " + to_string(numberOfCommand) + "\n";
    }
    else {
        identifer.type = Type::ARRAY;
        identiferTable.add(identifer.key, identifer.type);
    }

    arrayOnState = identifer;
    arrays.add(arrayOnState.key, 0);
    identifers.push(identifer);
    result.push_back(identifer);

    break;
}

case State::CAST_ARRAY_ASSIGMENT: {
    for (int i = 0; i < identifers.size(); i++) {
        auto identifer = identifers.top();
        identifers.pop();
        if (!initializedIdentifers.contains(identifer.key)) {
initializedIdentifers.add(identifer.key, identifer.type);
        }
    }

    if (arrays.contains(arrayOnState.key)) {
        arrays.remove(arrayOnState.key);
    }

    arrays.add(arrayOnState.key, numberOfElements);
    numberOfElements = 0;
    result.push_back(Token(Type::ASSIGMENT, "="));

    break;
}
```

```
case State::CAST_ARRAY_EXPRESSION: {
   auto rule = _table->rule();

   auto arrayInstance = _commandStack.getTokensByRule(rule).front();

   if (arrayInstance.type == Type::ARRAY) {
      if (!identiferTable.contains(arrayInstance.key)) {
         errorMessage += "syntax error: undeclared variable {" +
arrayInstance.key + "} " + "in line " + to_string(numberOfCommand) + "\n";
      }
       if (!initializedIdentifers.contains(arrayInstance.key)) {
           errorMessage += "syntax error: uninitialized variable {" +
arrayInstance.key + "} " + "in line " + to_string(numberOfCommand) + "\n";
       }
   }

   arrayOnState = arrayInstance;

   result.push_back(arrayInstance);

   break;
}
```