Томский государственный университет систем управления и радиоэлектроники

Кафедра КИБЭВС

Методические указания для проведения практических и самостоятельных работ по дисциплине «Безопасность программного обеспечения»

Выполнил ассистент каф. КИБЭВС

Сарин К.С

Томск 2016

Практическая работа 1

Обфускация - приведение исполняемого кода к виду, сохраняющему функциональность программы, но затрудняющему анализ и понимание алгоритмов работы.

Методы обфускации

1 Лексическая обфускация

Данный вид обфускации заключается в форматировании кода программы, изменении его структуры, таким образом, чтобы он стал нечитабельным, менее информативным и трудным для изучения.

Лексическая обфускация включает в себя:

- удаление всех комментариев в коде программы, или изменение их на дезинформирующие;
- удаление различных пробелов, отступов которые обычно используют для лучшего визуального восприятия кода программы;
- замену имен идентификаторов (имен переменных, функций и т.д.), на произвольные длинные наборы символов;
- добавление различных лишних (мусорных) операций;
- изменение расположения блоков (функций, процедур).

2 Вычислительная обфускация

Изменение касающиеся главной структуры потока управления. К ним можно отнести:

- расширение условий циклов;
- добавление недостижимого кода;
- добавление избыточных операций.

Ход работы

1. Выбрать код программы по варианту (варианты в конце).
2. Скомпилировать программу, проверить работоспособность.

   2.1 Запустить MS Visual Studio.

   2.2 Создать проект Visual C++/Win32/Консольное приложение Win32 с именем «test» (рис.1).

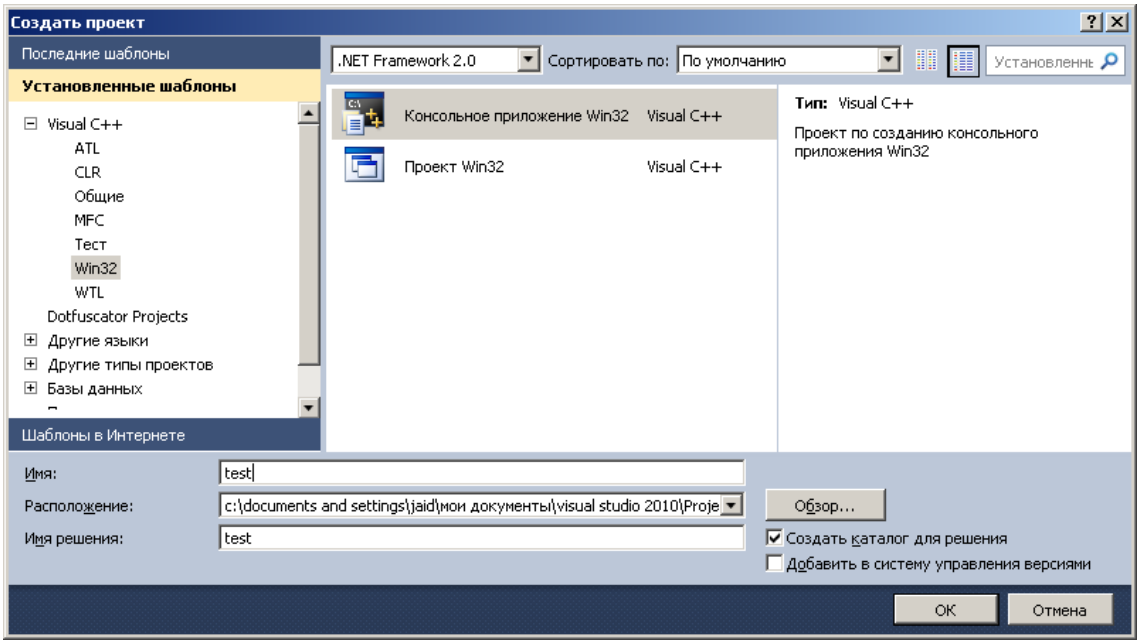


Рисунок 1. Новый проект в Visual Studio

2.3 Вставить код программы, полученный в пункте 1, в «test.cpp».

2.4 Нажать F5 для компиляции и запуска программы.

3. Осуществить лексическую обфускацию.

3.1 Заменить названия переменных на трудночитаемые, не несущие смысла.

До	После

`#include <stdafx.h>`
`#include <stdio.h>`
`// главная функция`
`int main()`
`{`
`// номер варианта`
`int q = 1;`
`switch( q )`
`{`
`// пусто (вариант 0)`
`case 0:`
`printf("null\n");`
`break;`
`// первый вариант`
`case 1:`
`for (int i=0;i<=1;i++)`
`printf("variant 1\n");`
`break;`
`// второй вариант`
`case 2:`
`printf("variant 2\n");`
	`#include <stdafx.h>`
	`#include <stdio.h>`
	`// главная функция`
	`int main()`
	`{`
	`// номер варианта`
	`int xX03123012 = 1;`
	`switch( xX03123012 )`
	`{`
	`// пусто (вариант 0)`
	`case 0:`
	`printf("null\n");`
	`break;`
	`// первый вариант`
	`case 1:`
	`for (int weq83efas_23=0;weq83efas_23<=1;weq83efas_23++)`
	`printf("variant 1\n");`
	`break;`
	`// второй вариант`

```
            break;
            // другие варианты
        default:
            printf("other variant\n");
        }
        scanf("%i",&q);
        return 0;
}
```

```
        case 2:
            printf("variant 2\n");
            break;
            // другие варианты
        default:
            printf("other variant\n");
        }
        scanf("%weq83efas_23",&xX03123012);
        return 0;
}
```

Комментарии к коду:

- заменено название переменной q на xX03123012;
- заменено название переменной i на weq83efas_23.

– 3.2 Использовать шестнадцатеричное представление чисел (пример приведен в описании лексической обфускации).

До

```
#include <stdafx.h>
#include <stdio.h>
// главная функция
int main()
{
        // номер варианта
        int xX03123012 = 1;
        switch( xX03123012 )
        {
            // пусто (вариант 0)
        case 0:
            printf("null\n");
            break;
            // первый вариант
        case 1:
            for (int
weq83efas_23=0;weq83efas_23<=1;weq83efas_
23++)
                printf("variant
1\n");
            break;
            // второй вариант
        case 2:
            printf("variant 2\n");
            break;
```

После

```
#include <stdafx.h>
#include <stdio.h>
// главная функция
int main()
{
        // номер варианта
        int xX03123012 = 0x26d2+60-0x270d;
        switch( xX03123012 )
        {
            // пусто (вариант 0)
        case 0x36d-877:
            printf("null\n");
            break;
            // первый вариант
        case -3550+0xddf:
            for (int
weq83efas_23=0;weq83efas_23<=1;weq83efas_23
++)
                printf("variant 1\n");
            break;
            // второй вариант
        case 0xca4-3233+1:
            printf("variant 2\n");
            break;
            // другие варианты
```

```
                // другие варианты
        default:
                printf("other variant\n");
        }
        scanf("%weq83efas_23",&xX03123012)
;
        return 0;
}
```

```
        default:
                printf("other variant\n");
        }
        scanf("%weq83efas_23",&xX03123012);
        return xX03123012-1;
}
```

– Комментарии к коду:

– применено шестнадцатеричное представление чисел, вместо десятичного (например, 0x26d2+60-0x270d=9938+60-9997=1, 2=0xca4-3233+1);

3.3 Убрать пробелы и отступы и заменить комментарии на дезинформирующие.

До

```
#include <stdafx.h>
#include <stdio.h>
// главная функция
int main()
{
        // номер варианта
        int xX03123012 = 0x26d2+60-0x270d;
        switch( xX03123012 )
        {
                // пусто (вариант 0)
        case 0x36d-877:
                printf("null\n");
                break;
                // первый вариант
        case -3550+0xddf:
                for (int
weq83efas_23=0;weq83efas_23<=1;weq83efas_
23++)
                        printf("variant
1\n");
                break;
                // второй вариант
        case 0xca4-3233+1:
                printf("variant 2\n");
                break;
```

После

```
// модуль распознавания речи
#include <stdafx.h>
#include <stdio.h>
int
main(){/*номер записи*/int
xX03123012=0x26d2+60-
0x270d;switch(xX03123012){/*голос
шпиона*/case 0x36d-
877:printf("null\n");break;/* голос
офицера*/case -3550+0xddf: for (int
weq83efas_23=0;weq83efas_23<=1;weq83efas_23
++)printf("variant 1\n");break;/*мой
голос*/case 0xca4-3233+1:printf("variant
2\n");break;/*другие
голоса*/default:printf("other
variant\n");}return xX03123012-1;}
```

```
                // другие варианты
        default:
                printf("other variant\n");
        }
        scanf("%weq83efas_23",&xX03123012)
;
        return xX03123012-1;
}
```

3.4 Скомпилировать программу клавишей F5 и проверить ее функциональность. При появлении ошибок необходимо восстановить код согласно варианту и повторить лексическую обфускацию.

3.5 Сделать резервную копию кода программы (например, в текстовом файле).

4. Осуществить вычислительную обфускацию.

4.1 Расширить условия циклов.

До

```
// модуль распознавания речи
#include <stdafx.h>
#include <stdio.h>
int
main(){/*номер записи*/int
xX03123012=0x26d2+60-
0x270d;switch(xX03123012){/*голос
шпиона*/case 0x36d-
877:printf("null\n");break;/* голос
офицера*/case -3550+0xddf: for (int
weq83efas_23=0;weq83efas_23<=1;weq83efas_
23++)printf("variant 1\n");break;/*мой
голос*/case 0xca4-3233+1:printf("variant
2\n");break;/*другие
голоса*/default:printf("other
variant\n");}return xX03123012-1;}
```

После

```
// модуль распознавания речи
#include <stdafx.h>
#include <stdio.h>
int
main(){/*номер записи*/int
xX03123012=0x26d2+60-
0x270d;switch(xX03123012){/*голос
шпиона*/case 0x36d-
877:printf("null\n");break;/* голос
офицера*/case -3550+0xddf: for (int
weq83efas_23=0, j33floor1=2;weq83efas_23<=1
&&
j33floor1==0x2;weq83efas_23++)printf("varia
nt 1\n");break;/*мой голос*/case 0xca4-
3233+1:printf("variant 2\n");break;/*другие
голоса*/default:printf("other
variant\n");}scanf
("%i",&xX03123012);return xX03123012-1;}
```

Комментарии к коду:

– в цикл for была включена переменная j33floor1, равная двум всегда, то есть, добавленное условие j33floor1==0x2 будет всегда выполняться.

4.2 Добавить избыточные операции.

До	После

// модуль распознавания речи
#include <stdafx.h>
#include <stdio.h>
int
main(){/*номер записи*/int
xX03123012=0x26d2+60-
0x270d;switch(xX03123012){/*голос
шпиона*/case 0x36d-
877:printf("null\n");break;/* голос
офицера*/case -3550+0xddf: for (int
weq83efas_23=0,
j33floor1=2;weq83efas_23<=1 &&
j33floor1==0x2;weq83efas_23++)printf("var
iant 1\n");break;/*мой голос*/case 0xca4-
3233+1:printf("variant
2\n");break;/*другие
голоса*/default:printf("other
variant\n");}scanf
("%i",&xX03123012);return xX03123012-1;}
	// модуль распознавания речи
	#include <stdafx.h>
	#include <stdio.h>
	int
	main(){/*номер записи*/int
	xX03123012=0x26d2+60-
	0x270d;switch(xX03123012){/*голос
	шпиона*/case 0x36d-
	877:printf("null\n");break;/* голос
	офицера*/case -3550+0xddf:for (int
	weq83efas_23=0, j33floor1=2;
	weq83efas_23<=1 && j33floor1==0x2;
	weq83efas_23++){ j33floor1=
	j33floor1*(21+0x4)/(15+0xa);printf("variant
	1\n");}break;/*мой голос*/case 0xca4-
	3233+1:printf("variant 2\n");break;/*другие
	голоса*/default:printf("other
	variant\n");}scanf
	("%i",&xX03123012);return xX03123012-1;}

Комментарии к коду:

– добавлена избыточная операция j33floor1=j33floor1*(21+0x4)/(15+0xa) (эта операция никак не изменяет значение j33floor1, так как множитель всегда равен единице).

4.3 Добавить недостижимый код.

До	После
// модуль распознавания речи #include <stdafx.h> #include <stdio.h> int main(){/*номер записи*/int xX03123012=0x26d2+60- 0x270d;switch(xX03123012){/*голос шпиона*/case 0x36d-	// модуль распознавания речи #include <stdafx.h> #include <stdio.h> int main(){/*номер записи*/int xX03123012=0x26d2+60- 0x270d;switch(xX03123012){/*голос шпиона*/case 0x36d-

877:printf("null\n");break;/* голос офицера*/case -3550+0xddf:for (int weq83efas_23=0, j33floor1=2; weq83efas_23<=1 && j33floor1==0x2; weq83efas_23++){ j33floor1= j33floor1*(21+0x4)/(15+0xa); printf("variant 1\n");}break;/*мой голос*/case 0xca4-3233+1:printf("variant 2\n");break;/*другие голоса*/default:printf("other variant\n");}scanf ("%i",&xX03123012);return xX03123012-1;}	877:printf("null\n");break;/* голос офицера*/case -3550+0xddf:for (int weq83efas_23=0, j33floor1=2; weq83efas_23<=1 && j33floor1==0x2; weq83efas_23++){ j33floor1= j33floor1*(21+0x4)/(15+0xa);if (j33floor1==0x13-2) weq83efas_23++;printf("variant 1\n");}break;/*мой голос*/case 0xca4-3233+1:printf("variant 2\n");break;/*другие голоса*/default:printf("other variant\n");}scanf ("%i",&xX03123012);return xX03123012-1;}

Комментарии к коду:

– добавлен недостижимый код в цикл if (j33floor1==0x13-2) weq83efas_23++ (условие никогда не выполнится, так как в данной программе j33floor1 равен 2 всегда).

4.4 Скомпилировать программу клавишей F5 и проверить ее функциональность. При появлении ошибок необходимо восстановить код из резервной копии и повторить вычислительную обфускацию.

4.5 Сохранить проект.

5. Поменяться обфусцированным кодом с одногруппником.
6. Попытаться разобраться в коде программы, полученном от одногруппника, и привести код к виду, близкому к исходному (под исходным видом понимается читабельный код с пробелами и отступами, без мусорных операций, с вашими подробными комментариями того, что делает программа).
7. Скомпилировать программу. Убедиться, что программа выполняет требуемые функции. В случае ошибок при компиляции, повторить пункт 6.
8. Ответить на вопросы.

Вопросы:

1. Что такое обфускация?
2. Зачем применяют обфускацию?
3. Какие методы обфускации рассмотрены в данной работе?
4. В чем заключается лексическая обфускация?
5. В чем заключается вычисительная обфускация?
6. Есть обычный код программы и обфусцированный код этой же программы. Каждый код скомпилировали и создали исполняемые файлы программ. Как вы думаете, какая программа потребует больше вычислительных ресурсов при выполнении? Обоснуйте свой ответ.

Варианты:

1)

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
 int X;
 cout<<"Vvedite cifru: ";
 cin>>X;
 // По-шаговые вычесления
 for(int i(1) ; i < 10 ; i++)
 {
        X = X * i;
        cout<<endl<<"Resultat: "<<X;

 }
 cin>>X;
}
```

2)

```
#include <stdafx.h>
#include<iostream>
#include<conio.h>

using std::cout;
using std::endl;
using std::cin;

const int n = 5;

void main()
{
 int mas[n];

 cout<<"Vvedite masiv: \n";
 for (int i = 0; i<n; i++)
 {
        cin>>mas[i];
 }

 int i = 0;

 do{
        mas[i] = mas[i]+5;
        i++;
 }while(i<5);
 cout<<"\nMasiv posle dobavleniya:\n";
 for (i = 0; i<n; i++)
 {
        cout<<mas[i]<<"\t";
 }

 _getch();
}
```

3)

```
#include <stdafx.h>
#include <iostream>
using namespace std;

int main()
{
 int i; // счетчик цикла
 int sum = 0; // сумма чисел от 1 до 1000.
 setlocale(0, "");
 for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и
задаем шаг цикла - 1.
 {
        sum = sum + i;
 }
 cout << "Сумма чисел " << sum << endl;
 cin >> i;
 return 0;
}
```

4)

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
 int speed = 5, count = 1;
 while ( speed < 60 )
 {
        speed += 10; // приращение скорости
        cout << count <<"-speed = " << speed << endl;
        count++; // подсчёт повторений цикла
 }
 system("pause");
 return 0;
}
```

5)

```
#include "stdafx.h"
#include <iostream>
#include <ctime>
using namespace std;

int main(int argc, char* argv[])
{
 srand( time( 0 ) );
 int unknown_number = 1 + rand() % 10; // загадываемое число
 int enter_number; // переменная для хранения введённого числа
 cout << "Enter number [1:10] : "; // начинаем отгадывать
 cin >> enter_number;
 while ( enter_number != unknown_number  )
 {
        cout << "Enter number [1:10] : ";
        cin >> enter_number; // продолжаем отгадывать
 }
 cout << "You win!!!\n";
 system("pause");
 return 0;
}
```

6)

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
 int counter_even = 0;
 for (int count = 2; count <= 50; count += 2) // заголовок цикла
        counter_even ++; // подсчёт чётных чисел
 cout << "number of even numbers = " << counter_even << endl;
 system("pause");
 return 0;
}
```

7)

```
#include "stdafx.h"
#include <iostream>
#include <ctime>
using namespace std;

int main(int argc, char* argv[])
{
 srand(time(0));
 int balance = 8; // баланс
 do // начало цикла do while
 {
        cout << "balance = " << balance << endl; // показать баланс
        int removal = rand() % 3; // переменная, для хранения вычитаемого
значения
        cout << "removal = " << removal << endl; // показать вычитаемое
значение
        balance -= removal; // управление условием
 }
 while ( balance > 0 ); // конец цикла do while
 system("pause");
 return 0;
}
```

8)

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
 cout << "Enter the first limit: "; // начальное значение из интервала
 int first_limit;
 cin >> first_limit;
 cout << "Enter the second limit: "; // конечное значение из интервала
 int second_limit;
 cin >> second_limit;
 int sum = 0, count = first_limit;
 do
 {
        sum += count; // наращивание суммы
        count++; // инкремент начального значения из задаваемого интервала
 } while (count <= second_limit); // конец цикла do while
 cout << "sum = " << sum << endl; // печать суммы
 system("pause");     return 0;}
```

Лабораторная работа 2

Дебфускация – процесс обратный обфускации, т.е. приведение кода к виду, близкому к исходному. Деобфускация предполагает оптимизацию кода.

В процессе обфускации в программный код часто производиться добавление лишних операций, они обычно никоим образом не влияют на результаты работы самой программы, и предназначены для усложнения процесса изучения кода программы потусторонними лицами.

В свою очередь процесс оптимизации программного кода направлен на ликвидацию лишних операций, поэтому в частных случаях он может выступать в качестве квинтэссенции процесса деобфускации.

Полностью восстановить первоначальный код не удастся, так как названия переменных, функций, классов и других объектов программы обычно изменяются, а узнать что-либо об их первоначальных значениях не представляется возможным.

Ход работы

1. Поменяться с одногруппником обфусцированным кодом, полученным в результате выполнения первой лабораторной работы.
2. Попытаться разобраться в коде программы, полученном от одногруппника, и привести код к виду, близкому к исходному (под исходным видом понимается читабельный код с пробелами и отступами, без мусорных операций, с вашими подробными комментариями того, что делает программа).

   2.1 Удалить все комментарии, так они могут быть изменены на дезинформирующие. Расставить отступы, пробелы табуляции.

Полученный код

```
// модуль распознавания речи
#include <stdafx.h>
#include <stdio.h>
int
main(){/*номер записи*/int xX03123012=0x26d2+60-
0x270d;switch(xX03123012){/*голос шпиона*/case 0x36d-
877:printf("null\n");break;/* голос офицера*/case -3550+0xddf:for (int
weq83efas_23=0, j33floor1=2; weq83efas_23<=1 && j33floor1==0x2; weq83efas_23++){
j33floor1= j33floor1*(21+0x4)/(15+0xa);if (j33floor1==0x13-2)
```

```
weq83efas_23++;printf("variant 1\n");}break;/*мой голос*/case 0xca4-
3233+1:printf("variant 2\n");break;/*другие голоса*/default:printf("other
variant\n");}scanf ("%i",&xX03123012);return xX03123012-1;}
```

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
	int xX03123012=0x26d2+60-0x270d;
	switch(xX03123012)
	{
	case 0x36d-877:
		printf("null\n");
		break;
	case -3550+0xddf:
		for (int weq83efas_23=0, j33floor1=2; weq83efas_23<=1 &&
j33floor1==0x2; weq83efas_23++)
		{
			j33floor1= j33floor1*(21+0x4)/(15+0xa);
			if (j33floor1==0x13-2) weq83efas_23++;
			printf("variant 1\n");
		}
		break;
	case 0xca4-3233+1:
		printf("variant 2\n");
		break;
	default:
		printf("other variant\n");
	}
	scanf ("%i",&xX03123012);
	return xX03123012-1;
}
```

2.2 Заменить имена переменных, классов, функций на более привычные Вам.

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
```

```
	int var1=0x26d2+60-0x270d;
	switch(var1)
	{
	case 0x36d-877:
		printf("null\n");
		break;
	case -3550+0xddf:
		for (int var2=0, var3=2; var2<=1 && var3==0x2; var2++)
		{
			var3= var3*(21+0x4)/(15+0xa);
			if (var3==0x13-2) var2++;
			printf("variant 1\n");
		}
		break;
	case 0xca4-3233+1:
		printf("variant 2\n");
		break;
	default:
		printf("other variant\n");
	}
	scanf ("%i",&var1);
	return var1-1;
}
```

2.3 Вычислить значения выражений и заменить их полученными константами.

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
	int var1=1;
	switch(var1)
	{
	case 0:
		printf("null\n");
		break;
	case 1:
		for (int var2=0, var3=2; var2<=1 && var3==2; var2++)
		{
			var3= var3*25/25;
			if (var3==17) var2++;
```

```
                    printf("variant 1\n");
                }
                break;
        case 2:
                printf("variant 2\n");
                break;
        default:
                printf("other variant\n");
        }
        scanf ("%i",&var1);
        return var1-1;
}
```

2.4 Упростить циклы за счет исключения расширенных условий, не влияющий на функционирование цикла. Заметим, что в условии цикла for переменная var3 сравнивается с «2». Перед выполнением цикла в var3 заносится значение «2», а в теле цикла эта переменная меняет значение на var3*25/25, то есть значение неизменно. Далее происходит сравнение «if (var3==17)», которое никогда не выполняется, ввиду того, что var3 всегда равно «2». Делаем вывод, что сравнение «var3==2» в условной части оператора лишнее, а переменная var3 не влияет на выполнение цикла, а ее значение нигде больше не используется. Поэтому, ее можно исключить из цикла.

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
        int var1=1;
        switch(var1)
        {
        case 0:
                printf("null\n");
                break;
        case 1:
                for (int var2=0; var2<=1; var2++)
                {
```

```
                printf("variant 1\n");
        }
        break;
    case 2:
        printf("variant 2\n");
        break;
    default:
        printf("other variant\n");
    }
    scanf ("%i",&var1);
    return var1-1;
}
```

2.5 Если просмотреть код, то можно обнаружить, что значение var1 всегда равно «1». Следовательно, можно исключить выражение «var1-1» в последней строке, заменив его «0».

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
    int var1=1;
    switch(var1)
    {
    case 0:
        printf("null\n");
        break;
    case 1:
        for (int var2=0; var2<=1; var2++)
        {

            printf("variant 1\n");
        }
        break;
    case 2:
        printf("variant 2\n");
        break;
    default:
        printf("other variant\n");
    }
```

```
        scanf ("%i",&var1);
        return 0;
}
```

3. Скомпилировать программу, проверить работоспособность (см. лаб. раб. 1).
4. Разобраться в том, что делает данная программа. Добавить ваши комментарии к коду, поясняющие основные моменты.

Результат

```
#include <stdafx.h>
#include <stdio.h>
//главная функция программы
int main()
{
        //инициализация переменной, от которой зависит выбор ветки switch
        int var1=1;
        //выбор варианта вывода на экран
        switch(var1)
        {
        case 0:
                //вывод фразы null и выход из оператора switch
                printf("null\n");
                break;
        case 1:
                //цикл двух повторений вывода на экран "variant 1"
                for (int var2=0; var2<=1; var2++)
                {
                        printf("variant 1\n");
                }
                break;
        case 2:
                //вывод на экран "variant 2"
                printf("variant 2\n");
                break;
        default:
                //вывод на экран "other variant"
                printf("other variant\n");
        }
        //задержка вывода
        scanf ("%i",&var1);
        return var0;
}
```

5. Ответить на вопросы

Вопросы:
1. Что такое деобфускация?
2. Зачем применяют деобфускацию?
3. С какими проблемами вы столкнулись при деобфускации кода?
4. Что невозможно восстановить при деобфускации кода?

Практическая работа 2

Деобфускация – процесс обратный обфускации, т.е. приведение кода к виду, близкому к исходному. Деобфускация предполагает оптимизацию кода.

В процессе обфускации в программный код часто производиться добавление лишних операций, они обычно никоим образом не влияют на результаты работы самой программы, и предназначены для усложнения процесса изучения кода программы потусторонними лицами.

В свою очередь процесс оптимизации программного кода направлен на ликвидацию лишних операций, поэтому в частных случаях он может выступать в качестве квинтэссенции процесса деобфускации.

Полностью восстановить первоначальный код не удастся, так как названия переменных, функций, классов и других объектов программы обычно изменяются, а узнать что-либо об их первоначальных значениях не представляется возможным.

Ход работы

1. Поменяться с одногруппником обфусцированным кодом, полученным в результате выполнения первой лабораторной работы.
2. Попытаться разобраться в коде программы, полученном от одногруппника, и привести код к виду, близкому к исходному (под исходным видом понимается читабельный код с пробелами и отступами, без мусорных операций, с вашими подробными комментариями того, что делает программа).

   2.1 Удалить все комментарии, так они могут быть изменены на дезинформирующие. Расставить отступы, пробелы табуляции.

Полученный код

```
// модуль распознавания речи
#include <stdafx.h>
#include <stdio.h>
int
main(){/*номер записи*/int xX03123012=0x26d2+60-
0x270d;switch(xX03123012){/*голос шпиона*/case 0x36d-
877:printf("null\n");break;/* голос офицера*/case -3550+0xddf:for (int
weq83efas_23=0, j33floor1=2; weq83efas_23<=1 && j33floor1==0x2; weq83efas_23++){
j33floor1= j33floor1*(21+0x4)/(15+0xa);if (j33floor1==0x13-2)
weq83efas_23++;printf("variant 1\n");}break;/*мой голос*/case 0xca4-
```

```
3233+1:printf("variant 2\n");break;/*другие голоса*/default:printf("other
variant\n");}scanf ("%i",&xX03123012);return xX03123012-1;}
```

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
	int xX03123012=0x26d2+60-0x270d;
	switch(xX03123012)
	{
	case 0x36d-877:
		printf("null\n");
		break;
	case -3550+0xddf:
		for (int weq83efas_23=0, j33floor1=2; weq83efas_23<=1 &&
j33floor1==0x2; weq83efas_23++)
		{
			j33floor1= j33floor1*(21+0x4)/(15+0xa);
			if (j33floor1==0x13-2) weq83efas_23++;
			printf("variant 1\n");
		}
		break;
	case 0xca4-3233+1:
		printf("variant 2\n");
		break;
	default:
		printf("other variant\n");
	}
	scanf ("%i",&xX03123012);
	return xX03123012-1;
}
```

2.2 Заменить имена переменных, классов, функций на более привычные Вам.

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
	int var1=0x26d2+60-0x270d;
```

```
	switch(var1)
	{
	case 0x36d-877:
		printf("null\n");
		break;
	case -3550+0xddf:
		for (int var2=0, var3=2; var2<=1 && var3==0x2; var2++)
		{
			var3= var3*(21+0x4)/(15+0xa);
			if (var3==0x13-2) var2++;
			printf("variant 1\n");
		}
		break;
	case 0xca4-3233+1:
		printf("variant 2\n");
		break;
	default:
		printf("other variant\n");
	}
	scanf ("%i",&var1);
	return var1-1;
}
```

2.3 Вычислить значения выражений и заменить их полученными константами.

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
	int var1=1;
	switch(var1)
	{
	case 0:
		printf("null\n");
		break;
	case 1:
		for (int var2=0, var3=2; var2<=1 && var3==2; var2++)
		{
			var3= var3*25/25;
			if (var3==17) var2++;
			printf("variant 1\n");
```

```
                }
                break;
        case 2:
                printf("variant 2\n");
                break;
        default:
                printf("other variant\n");
        }
        scanf ("%i",&var1);
        return var1-1;
}
```

2.4 Упростить циклы за счет исключения расширенных условий, не влияющий на функционирование цикла. Заметим, что в условии цикла for переменная var3 сравнивается с «2». Перед выполнением цикла в var3 заносится значение «2», а в теле цикла эта переменная меняет значение на var3*25/25, то есть значение неизменно. Далее происходит сравнение «if (var3==17)», которое никогда не выполняется, ввиду того, что var3 всегда равно «2». Делаем вывод, что сравнение «var3==2» в условной части оператора лишнее, а переменная var3 не влияет на выполнение цикла, а ее значение нигде больше не используется. Поэтому, ее можно исключить из цикла.

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
        int var1=1;
        switch(var1)
        {
        case 0:
                printf("null\n");
                break;
        case 1:
                for (int var2=0; var2<=1; var2++)
                {
```

```
                        printf("variant 1\n");
                }
                break;
        case 2:
                printf("variant 2\n");
                break;
        default:
                printf("other variant\n");
        }
        scanf ("%i",&var1);
        return var1-1;
}
```

2.5 Если просмотреть код, то можно обнаружить, что значение var1 всегда равно «1». Следовательно, можно исключить выражение «var1-1» в последней строке, заменив его «0».

Результат

```
#include <stdafx.h>
#include <stdio.h>
int main()
{
        int var1=1;
        switch(var1)
        {
        case 0:
                printf("null\n");
                break;
        case 1:
                for (int var2=0; var2<=1; var2++)
                {

                        printf("variant 1\n");
                }
                break;
        case 2:
                printf("variant 2\n");
                break;
        default:
                printf("other variant\n");
        }
        scanf ("%i",&var1);
```

```
	return 0;
}
```

3. Скомпилировать программу, проверить работоспособность (см. лаб. раб. 1).
4. Разобраться в том, что делает данная программа. Добавить ваши комментарии к коду, поясняющие основные моменты.

Результат

```
#include <stdafx.h>
#include <stdio.h>
//главная функция программы
int main()
{
	//инициализация переменной, от которой зависит выбор ветки switch
	int var1=1;
	//выбор варианта вывода на экран
	switch(var1)
	{
	case 0:
		//вывод фразы null и выход из оператора switch
		printf("null\n");
		break;
	case 1:
		//цикл двух повторений вывода на экран "variant 1"
		for (int var2=0; var2<=1; var2++)
		{
			printf("variant 1\n");
		}
		break;
	case 2:
		//вывод на экран "variant 2"
		printf("variant 2\n");
		break;
	default:
		//вывод на экран "other variant"
		printf("other variant\n");
	}
	//задержка вывода
	scanf ("%i",&var1);
	return var0;
}
```

5. Ответить на вопросы

Вопросы:
1. Что такое деобфускация?
2. Зачем применяют деобфускацию?
3. С какими проблемами вы столкнулись при деобфускации кода?
4. Что невозможно восстановить при деобфускации кода?

Практическая работа 3

В данной лабораторной работе рассматривается возможность изучения исполняемых файлов программ с помощью дизассемблера IDA Pro.

Дизассемблер — транслятор, преобразующий машинный код, объектный файл или библиотечные модули в текст программы на языке ассемблера.

Чаще всего дизассемблер используют для анализа программы (или ее части), исходный текст которой неизвестен — с целью модификации, копирования или взлома. Реже — для поиска ошибок в программах и компиляторах, а также для анализа оптимизации создаваемого компилятором машинного кода.

Ход работы

1) Выбрать код программы по варианту (варианты в конце).
2) Скомпилировать программу, проверить работоспособность.
2.1) Запустить MS Visual Studio.
2.2) Создать проект Visual C++/Win32/Консольное приложение Win32 с именем «test» (рис.1).

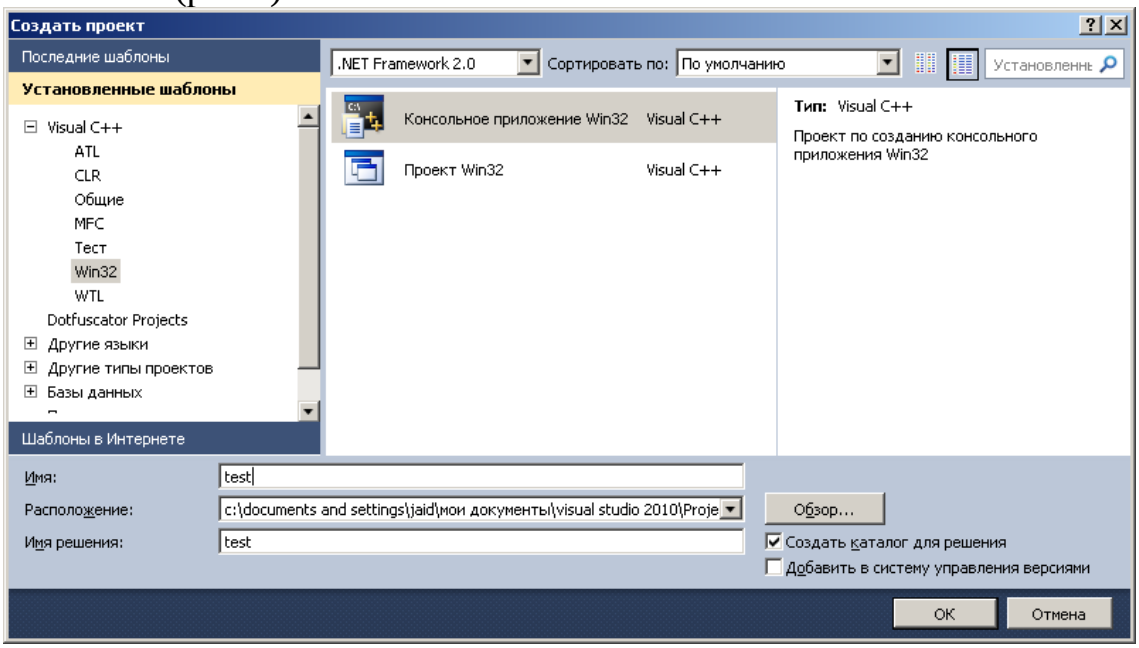


Рисунок 1. Новый проект в Visual Studio

2.3) Вставить код программы, полученный в пункте 1, в «test.cpp».
2.4) Нажать F5 для компиляции и запуска программы.
3) Запустить IDA Pro (C:\Program Files\IDA\idag.exe).
4) Выбрать «New» в появившемся меню. Далее на вкладке «Windows» выбрать PE executable (рис. 2).

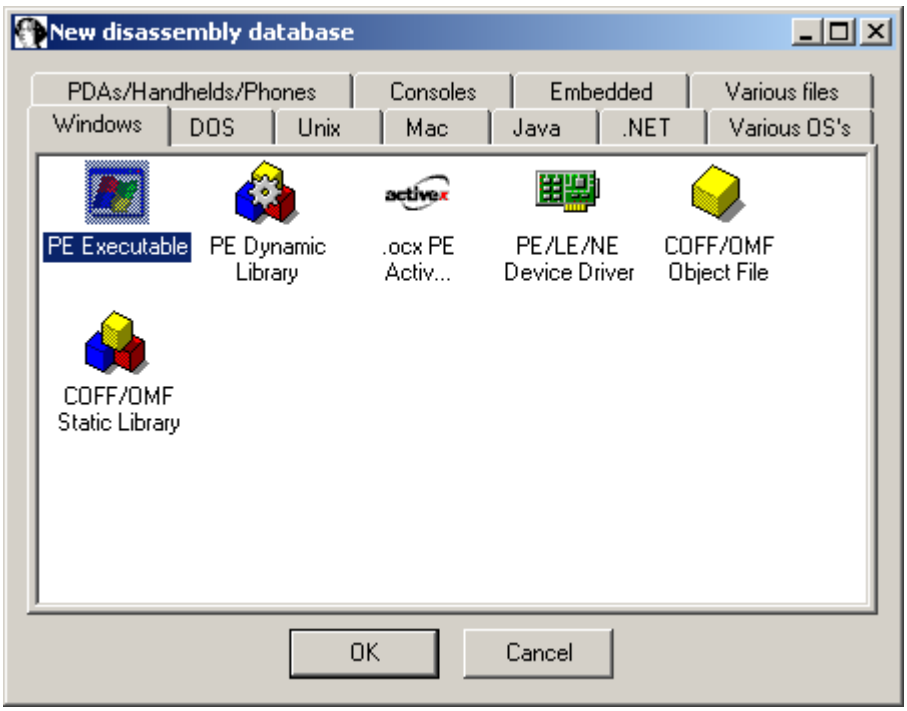


Рисунок 2. Объекты для дизассемблирования

5) В появившемся проводнике указать путь к исполняемому файлу вашей программы (Мои документы\visual studio 2010\Projects\ [папка с проектом]\Debug*.exe).

6) Нажимать Далее -> Далее -> Готово. Программа дизассемблирует исполняемый файл, выводит полученный код и графическую схему программы в окне Graph Overview. На верхней панели IDA Pro есть строка адресов, необходимо поставить желтую стрелочку на первую синюю полоску (рис. 3), после этого Graph Overview будет отображать схему функции main программы.

Рисунок 3. Адресная строка

Код программы для примера приведен ниже, его схема на рисунке 4.

```
#include <stdafx.h>
#include <stdio.h>
// главная функция
int main()
{
	// номер варианта
	int q = 1;
	switch( q )
	{
		// пусто (вариант 0)
	case 0:
		printf("null\n");
		break;
		// первый вариант
	case 1:
		for (int i=0;i<=1;i++)
			printf("variant 1\n");
		break;
		// второй вариант
	case 2:
```

```
            printf("variant 2\n");
            break;
            // другие варианты
        default:
            printf("other variant\n");
        }
        scanf("%i",&q);
        return 0;
}
```

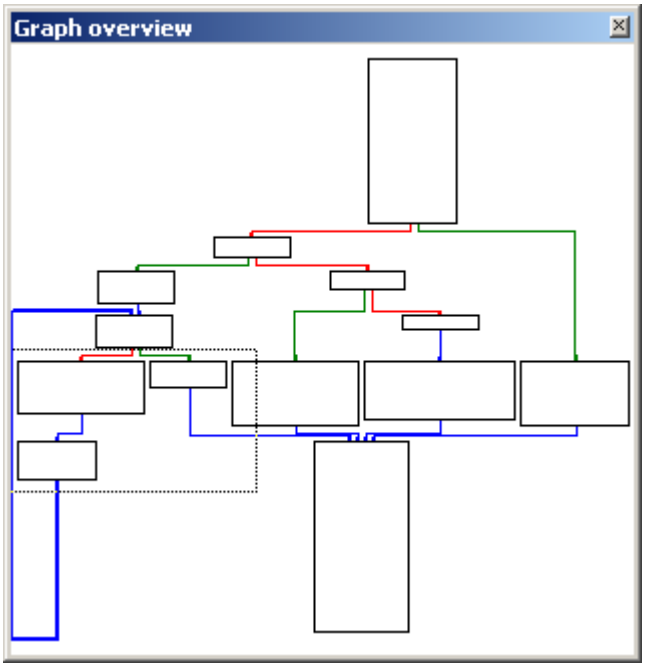


Рисунок 4. Схема программы

7) Проанализировать графическую схему. Найти циклы.

Рассмотрим краткий пример анализа дизассемблерного кода. Возьмем в качестве примера схему на рисунке 4. На ней можно отметить единственный цикл, стрелка возврата которого обозначена цифрой 1 (рис. 5). Также, просматривается оператор switch, представленный в виде условий if-else, обозначенный цифрой 2.

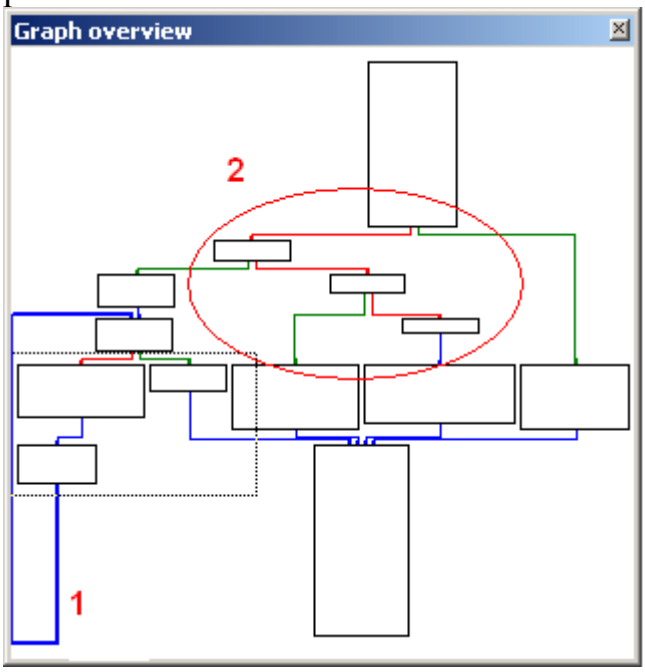


Рисунок 5. Разбор схемы

Если посмотреть на схему с кодом (рис. 6) области 2 рисунка 5, то можно представить, как работает программа.

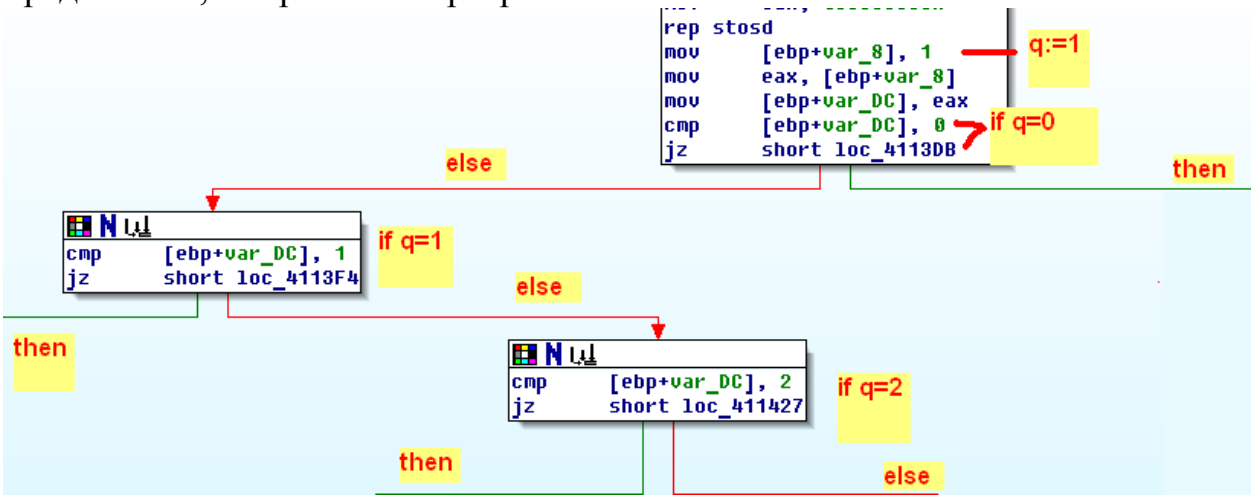


Рисунок 6. Схема с кодом оператора switch

Также можно рассмотреть отдельно цикл (рис. 7). Он в точности соответствует коду

```
for (int i=0;i<=1;i++)
        printf("variant 1\n").
```

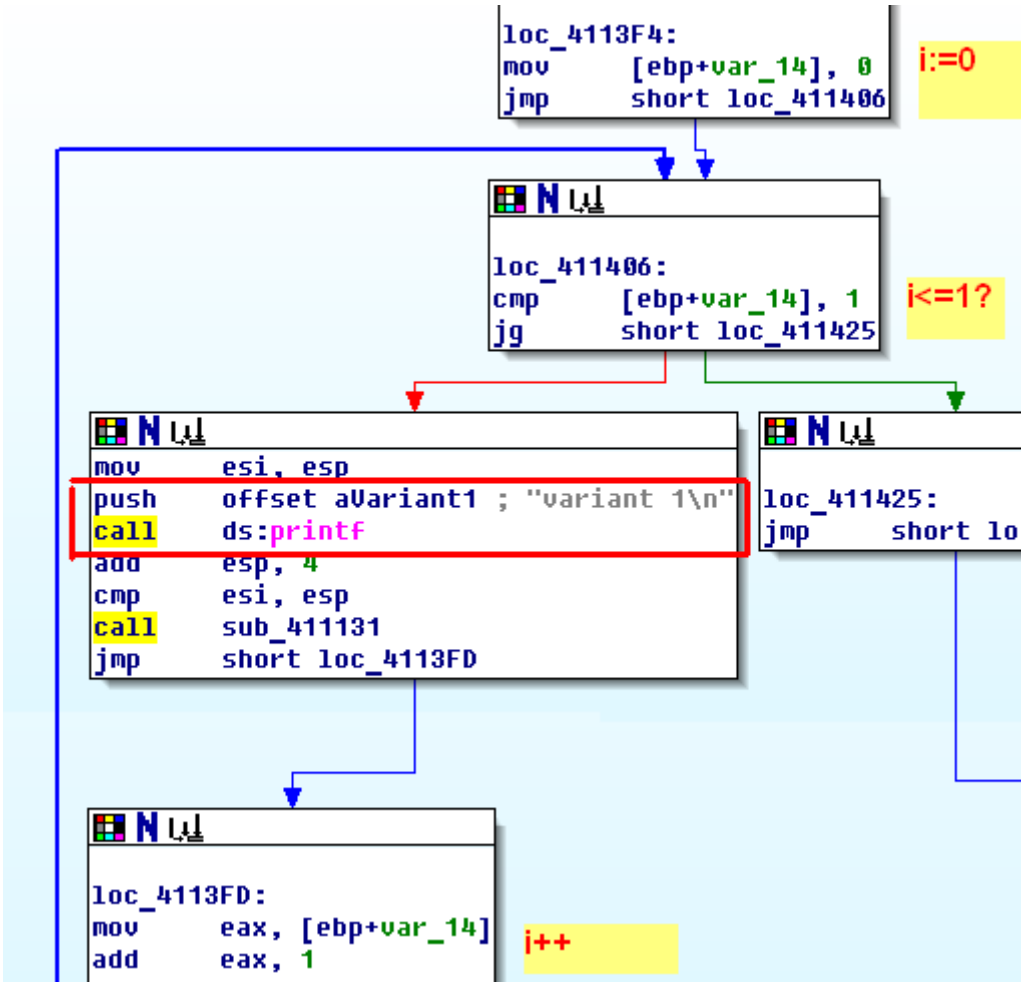


Рисунок 7. Схема с кодом оператора for

Все условия if-else сходятся в одном месте в самом низу схемы рисунка 5. Если посмотреть на схему с кодом, то можно наблюдать scanf () как раз в этом месте (рисунок 8).

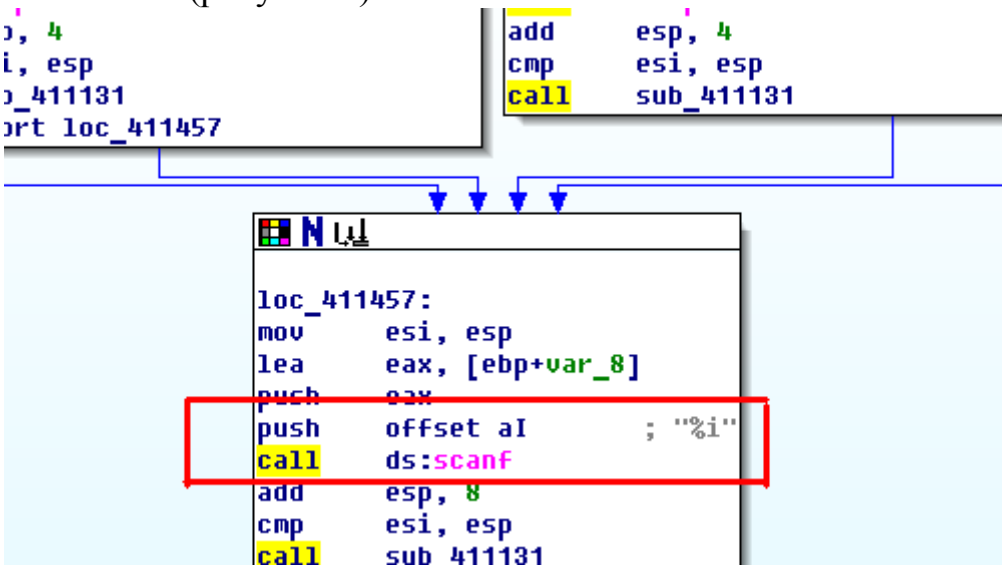


Рисунок 8. Конец программы

8) Дать оценку защищенности исполняемого файла и возможности его изучения.

9) Ответить на вопросы.

Вопросы:

1. Что такое дизассемблер?
2. Зачем используют дизассемблеры?
3. Что позволяет узнать дизассемблер?
4. Что не позволяет узнать дизассемблер?
5. Позволяет ли дизассемблер узнать исходный код программы? Обоснуйте ответ.

Варианты:

1)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
using namespace std;
void main()
{
	int X;
	cout<<"Vvedite cifru: ";
	cin>>X;
	// По-шаговые вычесления
	for(int i(1) ; i < 10 ; i++)
	{
		X = X * i;
		cout<<endl<<"Resultat: "<<X;
	}
	cin>>X;
}
```

2)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include<iostream>
#include<conio.h>

using std::cout;
using std::endl;
using std::cin;

const int n = 5;

void main()
{
	int mas[n];

	cout<<"Vvedite masiv: \n";
	for (int i = 0; i<n; i++)
	{
		cin>>mas[i];
	}

	int i = 0;

	do{
		mas[i] = mas[i]+5;
		i++;
	}while(i<5);
	cout<<"\nMasiv posle dobavleniya:\n";
	for (i = 0; i<n; i++)
	{
		cout<<mas[i]<<"\t";
	}

	_getch();
}
```

3)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
using namespace std;

int main()
{
	int i; // счетчик цикла
	int sum = 0; // сумма чисел от 1 до 1000.
	setlocale(0, "");
	for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и задаем шаг
цикла - 1.
	{
		sum = sum + i;
	}
	cout << "Сумма чисел " << sum << endl;
	cin >> i;
	return 0;
}
```

4)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
using namespace std;
```

```
int main(int argc, char* argv[])
{
	int speed = 5, count = 1;
	while ( speed < 60 )
	{
		speed += 10; // приращение скорости
		cout << count <<"-speed = " << speed << endl;
		count++; // подсчёт повторений цикла
	}
	system("pause");
	return 0;
}
```

5)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
#include <ctime>
using namespace std;

int main(int argc, char* argv[])
{
	srand( time( 0 ) );
	int unknown_number = 1 + rand() % 10; // загадываемое число
	int enter_number; // переменная для хранения введённого числа
	cout << "Enter number [1:10] : "; // начинаем отгадывать
	cin >> enter_number;
	while ( enter_number != unknown_number  )
	{
		cout << "Enter number [1:10] : ";
		cin >> enter_number; // продолжаем отгадывать
	}
	cout << "You win!!!\n";
	system("pause");
	return 0;
}
```

6)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
	int counter_even = 0;
	for (int count = 2; count <= 50; count += 2) // заголовок цикла
		counter_even ++; // подсчёт чётных чисел
	cout << "number of even numbers = " << counter_even << endl;
	system("pause");
	return 0;
}
```

7)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
#include <ctime>
using namespace std;

int main(int argc, char* argv[])
{
```

```
	srand(time(0));
	int balance = 8; // баланс
	do // начало цикла do while
	{
		cout << "balance = " << balance << endl; // показать баланс
		int removal = rand() % 3; // переменная, для хранения вычитаемого значения
		cout << "removal = " << removal << endl; // показать вычитаемое значение
		balance -= removal; // управление условием
	}
	while ( balance > 0 ); // конец цикла do while
	system("pause");
	return 0;
}
```

8)

```
#include <vcl.h>
#pragma hdrstop
#pragma argsused
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
	cout << "Enter the first limit: "; // начальное значение из интервала
	int first_limit;
	cin >> first_limit;
	cout << "Enter the second limit: "; // конечное значение из интервала
	int second_limit;
	cin >> second_limit;
	int sum = 0, count = first_limit;
	do
	{
		sum += count; // наращивание суммы
		count++; // инкремент начального значения из задаваемого интервала
	} while (count <= second_limit); // конец цикла do while
	cout << "sum = " << sum << endl; // печать суммы
	system("pause");
	return 0;
}
```

Практическая работа 4

Содержание

Введение

Дизассемблирование (От англ. disassemble - разбирать, демонтировать) – это процесс или способ получения исходного текста программы на ассемблере из программы в машинных кодах. Полезен при определении степени оптимальности транслятора и при генерации кодов собственной программы. Позволяет понять алгоритм или метод построения программ, у которых отсутствуют исходные тексты. Существуют специальные программы дизассемблеры, которые выполняют этот процесс.

Одним из передовых продуктов для дизассемблирования программ является пакет программ от CSO Computer Services - IDA (Interactive Disassembler). IDA не является автоматическим дизассемблером. Это означает, что IDA выполняет дизассемблирование лишь тех участков программного кода, на которые имеются явные ссылки. При этом IDA старается извлечь из кода максимум информации, не делая никаких излишних предположений. После завершения предварительного анализа программы, когда все обнаруженные явные ссылки исчерпаны, IDA останавливается и ждет вмешательства; просмотрев готовые участки текста, можно как бы подсказать ей, что нужно делать дальше. После каждого вмешательства снова запускается автоматический анализатор IDA, который на основе полученных сведений пытается продолжить дизассемблирование.

IDA является не только дизассемблером, но и одним из самых мощных средств исследования программ. Это возможно благодаря наличию развитой навигационной системы, позволяющей быстро перемещаться между различными точками программы, объектами и ссылками на них, отыскивать неявные ссылки и т.д. Исследование даже больших и сложных программ в IDA занимает в десятки и сотни раз меньше времени, чем путем просмотра текста, полученного обычным дизассемблером.

Целью, данной работы, является задача дизассемблирования программ написанных на языке программирования C/C++ и скомпилированных на компиляторах Microsoft Visual C++ 6.0, Borland C++ 5.0 и WATCOM.

В процесс дизассемблирования входит:

1) идентификация математических операций таких как: сложение, вычитание, деление, операция вычисления остатка, умножение и определения комплексных операций;

2) Идентификация операторов SWITCH - CASE – BREAK.

В ходе работы будет проведен анализ по качеству оптимизации, сгенерированных приведенными выше компиляторами программ, что позволит сравнить их и определить каким из них необходимо воспользоваться для получения наиболее оптимального и быстрого кода в программировании.

1. Идентификация математических операторов

1.1 Идентификация оператора "+"

В общем случае оператор "+" транслируется либо в машинную инструкцию ADD, "перемалывающую" целочисленные операнды, либо в инструкцию FADDx, обрабатывающую вещественные значения. Оптимизирующие компиляторы могут заменять "ADD xxx, 1" более компактной командой "INC xxx", а конструкцию "c = a + b + const" транслировать в машинную инструкцию "LEA c, [a + b + const]". Такой трюк позволяет одним махом складывать несколько переменных, возвратив полученную сумму в любом регистре общего назначения, - не обязательно в левом слагаемом как это требует мнемоника команды ADD. Однако, "LEA" не может быть непосредственно декомпилирована в оператор "+", поскольку она используется не только для оптимизированного сложения (что, в общем-то, побочный продукт ее деятельности), но и по своему непосредственному назначению - вычислению эффективного смещения. Рассмотрим следующий пример:

```
main()
{
int a, b,c;
c = a + b;
printf("%x\n",c);
c=c+1;
printf("%x\n",c);

}
```

Демонстрация оператора "+"

Результат его компиляции компилятором Microsoft Visual C++ 6.0 с настройками по умолчанию должен выглядеть так:

```
main            proc near               ; CODE XREF: start+AF p

var_c           = dword ptr -0Ch
var_b           = dword ptr -8
var_a           = dword ptr -4

push    ebp
mov     ebp, esp
; Открываем кадр стека

sub     esp, 0Ch
; Резервируем память для локальных переменных

mov     eax, [ebp+var_a]
; Загружаем в EAX значение переменной var_a
```

```
add     eax, [ebp+var_b]
; Складываем EAX со значением переменной var_b и записываем результат в EAX

mov     [ebp+var_c], eax
; Копируем сумму var_a и var_b в переменную var_c, следовательно:
; var_c = var_a + var_b

mov     ecx, [ebp+var_c]
push    ecx
push    offset asc_406030 ; "%x\n"
call    _printf
add     esp, 8
; printf("%x\n", var_c)

mov     edx, [ebp+var_c]
; Загружаем в EDX значение переменной var_c

add     edx, 1
; Складываем EDX со значением 0x1, записывая результат в EDX

mov     [ebp+var_c], edx
; Обновляем var_c
; var_c = var_c +1

mov     eax, [ebp+var_c]
push    eax
push    offset asc_406034 ; "%x\n"
call    _printf
add     esp, 8
; printf("%\n",var_c)

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn

main            endp
```

Теперь посмотрим, как будет выглядеть тот же самый пример, скомпилированный с ключом "/Ox" (максимальная оптимизация):

```
main            proc near               ; CODE XREF: start+AF p
push    ecx
; Резервируем место для одной локальной переменной
; (компилятор посчитал, что три переменные можно ужать в одну и это дейст. так)

mov     eax, [esp+0]
```

```
; Загружаем в EAX значение переменной var_a

mov     ecx, [esp+0]
; Загружаем в EAX значение переменной var_b
; (т.к .переменная не инициализирована загружать можно откуда угодно)

push    esi
; Сохраняем регистр ESI в стеке

lea     esi, [ecx+eax]
; Используем LEA для быстрого сложения ECX и EAX с последующей записью суммы
; в регистр ESI
; "Быстрое сложение" следует понимать не в смысле, что команда LEA выполняется
; быстрее чем ADD, - количество тактов той и другой одинаково, но LEA
; позволяет избавиться от создания временной переменной для сохранения
; промежуточного результата сложения, сразу направляя результат в ESI
; Таким образом, эта команда декомпилируется как
; reg_ESI = var_a + var_b

push    esi
push    offset asc_406030 ; "%x\n"
call    _printf
; printf("%x\n", reg_ESI)

inc     esi
; Увеличиваем ESI на единицу
; reg_ESI = reg_ESI + 1

push    esi
push    offset asc_406034 ; "%x\n"
call    _printf
add     esp, 10h
; printf("%x\n", reg_ESI)

pop     esi
pop     ecx
retn

main            endp
```

Остальные компиляторы (Borland C++, WATCOM C) генерируют приблизительно идентичный код, поэтому, приводить результаты бессмысленно - никаких новых "изюминок" они в себе не несут.

1.2 Идентификация оператора "-"

В общем случае оператор "- " транслируется либо в машинную инструкцию SUB (если операнды - целочисленные значения), либо в инструкцию FSUBx (если операнды - вещественные значения). Оптимизирующие компиляторы могут заменять "SUB xxx, 1" более компактной командой "DEC xxx", а конструкцию "SUB a, const" транслировать в "ADD a, -const", которая ничуть не компактнее и ни сколь не быстрей (и та, и другая укладываться в один так). Рассмотрим это в следующем примере:

```
main()
{
int a,b,c;

c = a - b;
printf("%x\n",c);

c = c - 10;
printf("%x\n",c);

}
```

Демонстрация идентификации оператора "-"

Не оптимизированный вариант будет выглядеть приблизительно так:

```
main            proc near               ; CODE XREF: start+AF p

var_c           = dword ptr -0Ch
var_b           = dword ptr -8
var_a           = dword ptr -4

push    ebp
mov     ebp, esp
; Открываем кадр стека

sub     esp, 0Ch
; Резервируем память под локальные переменные

mov     eax, [ebp+var_a]
; Загружаем в EAX значение переменной var_a

sub     eax, [ebp+var_b]
; Вычитаем из var_a значением переменной var_b, записывая результат в EAX

mov     [ebp+var_c], eax
; Записываем в var_c разность var_a и var_b
; var_c = var_a - var_b
```

```
mov     ecx, [ebp+var_c]
push    ecx
push    offset asc_406030 ; "%x\n"
call    _printf
add     esp, 8
; printf("%x\n", var_c)

mov     edx, [ebp+var_c]
; Загружаем в EDX значение переменной var_c

sub     edx, 0Ah
; Вычитаем из var_c значение 0xA, записывая результат в EDX

mov     [ebp+var_c], edx
; Обновляем var_c
; var_c = var_c - 0xA

mov     eax, [ebp+var_c]
push    eax
push    offset asc_406034 ; "%x\n"
call    _printf
add     esp, 8
; printf("%x\n",var_c)

mov     esp, ebp
pop     ebp
; Закрываем кадр стека
retn

main            endp
```

Теперь рассмотрим оптимизированный вариант того же примера:

```
main            proc near               ; CODE XREF: start+AF p
push    ecx
; Резервируем место для локальной переменной var_a

mov     eax, [esp+var_a]
; Загружаем в EAX значение локальной переменной var_a

push    esi
; Резервируем место для локальной переменной var_b

mov     esi, [esp+var_b]
; Загружаем в ESI значение переменной var_b

sub     esi, eax
; Вычитаем из var_a значение var_b, записывая результат в ESI
```

```
push    esi
push    offset asc_406030 ; "%x\n"
call    _printf
; printf("%x\n", var_a - var_b)

add     esi, 0FFFFFFF6h
; Добавляем к ESI (разности var_a и var_b) значение  0xFFFFFFF6
; Поскольку, 0xFFFFFFF6 == -0xA, данная строка кода выглядит так:
; ESI = (var_a - var_b) + (- 0xA) = (var_a - var_b) - 0xA

push    esi
push    offset asc_406034 ; "%x\n"
call    _printf
add     esp, 10h
; printf("%x\n", var_a - var_b - 0xA)

pop     esi
pop     ecx
; Закрываем кадр стека

retn

main            endp
```

Компиляторы (Borland, WATCOM) генерируют практически идентичный код.

1.3 Идентификация оператора "/"

В общем случае оператор "/" транслируется либо в машинную инструкцию "DIV" (беззнаковое целочисленное деление), либо в "IDIV" (целочисленное деление со знаком), либо в "FDIVx" (вещественное деление). Если делитель кратен степени двойки, то "DIV" заменяется на более быстродействующую инструкцию битового сдвига вправо "SHR a, N", где a - делимое, а N - показатель степени с основанием два.

Несколько сложнее происходит быстрое деление знаковых чисел. Совершенно недостаточно выполнить арифметический сдвиг вправо (команда арифметического сдвига вправо SAR заполняет старшие биты с учетом знака числа), ведь если модуль делимого меньше модуля делителя, то арифметический сдвиг вправо сбросит все значащие биты в "битовую корзину", в результате чего получиться 0xFFFFFFFF, т.е. -1, в то время как правильный ответ - ноль. Однако деление знаковых чисел арифметическим сдвигом вправо дает округление в **большую** сторону. Для округления знаковых чисел в меньшую сторону необходимо перед выполнением сдвига добавить к делимому число 2^N- 1, где N - количество битов, на которые сдвигается число при делении. Легко видеть, что это

приводит к увеличению всех сдвигаемых битов на единицу и переносу в старший разряд, если хотя бы один из них не равен нулю.

Следует отметить: деление очень медленная операция, гораздо более медленная чем умножение (выполнение DIV может занять свыше 40 тактов, в то время как MUL обычно укладываться в 4), поэтому, продвинутые оптимизирующие компиляторы заменяют деление умножением. Существует множество формул подобных преобразований, одной из самых популярных является: a/b = 2^N/b * a/2^N', где N' - разрядность числа. Следовательно, грань между умножением и делением очень тонка, а их идентификация является довольной сложной процедурой. Рассмотрим следующий пример:

```
main()
{
int a;
printf("%x %x\n",a / 32, a / 10);

}
```

Идентификация оператора "/"

Результат компиляции компилятором Microsoft Visual C++ с настройками по умолчанию должен выглядеть так:

```
main            proc near               ; CODE XREF: start+AF p

var_a           = dword ptr -4

push    ebp
mov     ebp, esp
; Открываем кадр стека

push    ecx
; Резервируем память для локальной переменной

mov     eax, [ebp+var_a]
; Копируем в EAX значение переменной var_a

cdq
; Расширяем EAX до четверного слова EDX:EAX

mov     ecx, 0Ah
; Заносим в ECX значение 0xA

idiv    ecx
; Делим (учитывая знак) EDX:EAX на 0xA, занося частное в EAX
; EAX = var_a / 0xA

push    eax
; Передаем результат вычислений функции printf
```

```
mov     eax, [ebp+var_a]
; Загружаем в EAX значение var_a

cdq
; Расширяем EAX до четверного слова EDX:EAX

and     edx, 1Fh
; Выделяем пять младших бит EDX

add     eax, edx
; Складываем знак числа для выполнения округления отрицательных значений
; в меньшую сторону

sar     eax, 5
; Арифметический сдвиг вправо на 5 позиций
; эквивалентен делению числа на 2^5 = 32
; Таким образом, последние четыре инструкции расшифровываются как:
; EAX = var_a / 32
; Обратите внимание: даже при выключенном режиме оптимизации компилятор
; оптимизировал деление

push    eax
push    offset aXX      ; "%x %x\n"
call    _printf
add     esp, 0Ch
; printf("%x %x\n", var_a / 0xA, var_a / 32)

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn

main            endp
```

Теперь, рассмотрим оптимизированный вариант того же примера:

```
main            proc near               ; CODE XREF: start+AF p
push    ecx
; Резервируем память для локальной переменной var_a

mov     ecx, [esp+var_a]
; Загружаем в ECX значение переменной var_a

mov     eax, 66666667h
; В исходном коде ничего подобного не было!
```

```
imul      ecx
; Умножаем это число на переменную var_a
; Обратите внимание: именно умножаем, а не делим.

sar       edx, 2
; Выполняем арифметический сдвиг всех битов EDX на две позиции вправо, что
; в первом приближении эквивалентно его делению на 4
; Однако ведь в EDX находятся старшее двойное слово результата умножения!
; Поэтому, три предыдущих команды фактически расшифровываются так:
; EDX = (66666667h * var_a) >> (32 + 2) = (66666667h * var_a) / 0x400000000
;
; Теперь немного упростим код:
; (66666667h * var_a) / 0x400000000 = var_a * 66666667h / 0x400000000 =
; = var_a * 0,10000000003492459654808044433594
; Заменяя по всем правилам математики умножение на деление и одновременно
; выполняя округление до меньшего целого получаем:
; var_a * 0,1000000000 = var_a * (1/0,1000000000) = var_a/10
;
; От такого преобразования код стал намного понятнее!
; Тогда возникает вопрос можно ли распознать такую ситуацию в чужой
; программе, исходный текст которой
; неизвестен? Можно - если встречается умножение, а следом за ним
; сдвиг вправо, обозначающий деление, сократив код, по методике показанной
выше!

mov       eax, edx
; Копируем полученное частное в EAX

shr       eax, 1Fh
; Сдвигаем на 31 позицию вправо

add       edx, eax
; Складываем: EDX = EDX + (EDX >> 31)
; Нетрудно понять, что после сдвига EDX на 31 бит вправо
; в нем останется лишь знаковый бит числа
; Тогда - если число отрицательно, добавляем к результату деления один,
; округляя его в меньшую сторону. Таким образом, весь этот хитрый код
; обозначает ни что иное как тривиальную операцию знакового деления:
; EDX = var_a / 10
; Конечно, программа становится очень громоздкой,
; зато весь этот код выполняется всего лишь за 9 тактов,
; в то время как в не оптимизированном варианте за 28!
; /* Измерения проводились на процессоре CLERION с ядром P6, на других
; процессорах количество тактов может отличается */
; Т.е. оптимизация дала более чем трехкратный выигрыш!

mov       eax, ecx
```

```
; Теперь нужно вспомнить: что находится в ECX.
; В ECX последний раз разгружалось значение переменной var_a

push    edx
; Передаем функции printf результат деления var_a на 10

cdq
; Расширяем EAX (var_a) до четверного слова EDX:EAX

and     edx, 1Fh
; Выбираем младшие 5 бит регистра EDX, содержащие знак var_a

add     eax, edx
; Округляем до меньшего

sar     eax, 5
; Арифметический сдвиг на 5 эквивалентен делению var_a на 32

push    eax
push    offset aXX      ; "%x %x\n"
call    _printf
add     esp, 10h
; printf("%x %x\n", var_a / 10, var_a / 32)

retn

main            endp
```

Однако такие компиляторы как Borland и WATCOM не умеют заменять деление более быстрым умножением для чисел отличных от степени двойки. В подтверждении тому рассмотрим результат компиляции того же примера компилятором Borland C++:

```
_main           proc near               ; DATA XREF: DATA:00407044 o

push    ebp
mov     ebp, esp
; Открываем кадр стека

push    ebx
; Сохраняем EBX

mov     eax, ecx
; Копируем в EAX содержимое неинициализированной регистровой переменной ECX

mov     ebx, 0Ah
```

```
; Заносим в EBX значение 0xA

cdq
; Расширяем EAX до четверного слова EDX:EAX

idiv    ebx
; Делим ECX на 0xA (примерно 20 тактов)

push    eax
; Передаем полученное значение функции printf

test    ecx, ecx
jns     short loc_401092
; Если делимое не отрицательно, то переход на loc_401092

add     ecx, 1Fh
; Если делимое положительно, то добавляем к нему 0x1F для округления

loc_401092:                             ; CODE XREF: _main+11 j
sar     ecx, 5
; Сдвигом на пять позиций вправо делим число на 32

push    ecx
push    offset aXX          ; "%x %x\n"
call    _printf
add     esp, 0Ch
; printf("%x %x\n", var_a / 10, var_a / 32)

xor     eax, eax
; Возвращаем ноль

pop     ebx
pop     ebp
; Закрываем кадр стека

retn

_main               endp
```

1.4 Идентификация оператора "%"

Специальной инструкции для вычисления остатка в наборе команд микропроцессоров серии 80x86 нет, - вместо этого остаток вместе с частным возвращается инструкциями деления DIV, IDIV и FDIVx. Если делитель представляет собой степень двойки (2^N = b), а делимое беззнаковое число, то остаток будет равен N младшим битам делимого числа. Если же делимое - знаковое, необходимо установить все биты, кроме первых N равными знаковому

биту для сохранения знака числа. Причем, если N первых битов равно нулю, все биты результата должны быть сброшены независимо от значения знакового бита.

Таким образом, если делимое - беззнаковое число, то выражение a % 2^N транслируется в конструкцию: "AND a, N", в противном случае трансляция становится неоднозначна - компилятор может вставлять явную проверку на равенство нулю с ветвлением, а может использовать хитрые математические алгоритмы, самый популярный из которых выглядит так: DEC x\ OR x, -N\ INC x. Весь фокус в том, что если первые N бит числа x равны нулю, то все биты результата кроме старшего, знакового бита, будут гарантированно равны одному, а OR x, -N принудительно установит в единицу и старший бит, т.е. получится значение, равное, -1. А INC -1 даст ноль! Напротив, если хотя бы один из N младших битов равен одному, заема из старших битов не происходит и INC x возвращает значению первоначальный результат.

Продвинутые оптимизирующие компиляторы могут путем сложных преобразований заменять деление на ряд других, более быстродействующих операций. К сожалению, алгоритмов для быстрого вычисления остатка для всех делителей не существует и делитель должен быть кратен k * 2^t, где k и t - некоторые целые числа. Тогда остаток можно вычислить по следующей формуле: a % b = a % k*2^t = a -((2^N/k * a/2^N) & -2^t)*k

Эта формула очень сложна и идентификация оптимизированного оператора "%" может быть весьма и весьма непростой, особенно учитывая, что оптимизаторы изменяют порядок команд.

Рассмотрим следующий пример:

```
main()
{

int a;
printf("%x %x\n",a % 16, a % 10);

}
```

Идентификация оператора "%"

Результат его компиляции компилятором Microsoft Visual C++ с настройками по умолчанию должен выглядеть так:

```
main            proc near               ; CODE XREF: start+AF p

var_4           = dword ptr -4
push    ebp
mov     ebp, esp
; Открываем кадр стека

push    ecx
; Резервируем память для локальной переменной

mov     eax, [ebp+var_a]
; Заносим в EAX значение переменной var_a
```

```
cdq
; Расширяем EAX до четвертного слова EDX:EAX

mov     ecx, 0Ah
; Заносим в ECX значение 0xA

idiv    ecx
; Делим EDX:EAX (var_a) на ECX (0xA)


push    edx
; Передаем остаток от деления var_a на 0xA функции printf

mov     edx, [ebp+var_a]
; Заносим в EDX значение переменной var_a

and     edx, 8000000Fh
; "Вырезаем" знаковый бит и четыре младших бита числа
; в четырех младших битах содержится остаток от деления EDX на 16

jns     short loc_401020
; Если число не отрицательно, то прыгаем на loc_401020

dec     edx
or      edx, 0FFFFFFF0h
inc     edx
; Эта последовательность, как говорилось выше, характера для быстрого
; расчета остатка знакового числа
; Следовательно, последние шесть инструкций расшифровываются как:
; EDX = var_a % 16

loc_401020:                             ; CODE XREF: main+19 j
push    edx
push    offset aXX          ; "%x %x\n"
call    _printf
add     esp, 0Ch
; printf("%x %x\n",var_a % 0xA, var_a % 16)

mov     esp, ebp
pop     ebp
; Закрываем кадр стека
retn

main            endp
```

Любопытно, что оптимизация не влияет на алгоритм вычисления остатка. Увы, ни Microsoft Visual C++, ни остальные известные компиляторы не умеют вычислять остаток умножением.

1.5 Идентификация оператора "*"

В общем случае оператор "*" транслируется либо в машинную инструкцию "MUL" (беззнаковое целочисленное умножение), либо в "IMUL" (целочисленное умножение со знаком), либо в "FMULx" (вещественное умножение). Если один из множителей кратен степени двойки, то "MUL" ("IMUL") обычно заменяется командой битового сдвига влево "SHL" или инструкцией "LEA", способной умножать содержимое регистров на 2, 4 и 8. Обе последних команды выполняются за один такт, в то время как MUL требует в зависимости от модели процессора от двух до девяти тактов. К тому же LEA за тот же такт успевает сложить результат умножение с содержимым регистра общего назначения и/или константой. Это позволяет умножать на 3, 5 и 9 просто добавляя к умножаемому регистру его значение. Правда, у операции LEA есть один недочет - она может вызывать остановку AGI, в конечном счете весь выигрыш в быстродействии сводится на нет.

Рассмотрим следующий пример:

```
main()
{

int a;
printf("%x %x %x\n",a * 16, a * 4 + 5, a * 13);

}
```

Идентификация оператора ""*

Результат его компиляции компилятором Microsoft Visual C++ с настройками по умолчанию должен выглядеть так:

```
main            proc near               ; CODE XREF: start+AF p

var_a           = dword ptr -4

push    ebp
mov     ebp, esp
; Открываем кадр стека

push    ecx
; Резервируем место для локальной переменной var_a

mov     eax, [ebp+var_a]
; Загружаем в EAX значение переменной var_a

imul    eax, 0Dh
; Умножаем var_a на 0xD, записывая результат в EAX
```

```
push    eax
; Передаем функции printf произведение var_a * 0xD

mov     ecx, [ebp+var_a]
; Загружаем в ECX значение var_a

lea     edx, ds:5[ecx*4]
; Умножаем ECX на 4 и добавляем к полученному результату 5, записывая его в EDX
; И все это выполняется за один такт!

push    edx
; Передаем функции printf результат var_a * 4 + 5

mov     eax, [ebp+var_a]
; Загружаем в EAX значение переменной var_a

shl     eax, 4
; Умножаем var_a на 16

push    eax
; Передаем функции printf произведение var_a * 16

push    offset aXXX     ; "%x %x %x\n"
call    _printf
add     esp, 10h
; printf("%x %x %x\n", var_a * 16, var_a * 4 + 5, var_a * 0xD)

mov     esp, ebp
pop     ebp
; Закрываем кадр стека
retn

main            endp
```

За вычетом вызова функции printf и загрузки переменной var_a из памяти уходит всего лишь *три* такта процессора. Если скомпилировать этот пример с ключом "/Ox", то получится вот что:

```
main            proc near               ; CODE XREF: start+AF p
push    ecx
; Выделяем память для локальной переменной var_a

mov     eax, [esp+var_a]
; Загружаем в EAX значение переменной var_a
```

```
lea         ecx, [eax+eax*2]
; ECX = var_a * 2 + var_a = var_a * 3

lea         edx, [eax+ecx*4]
; EDX = (var_a * 3)* 4 + var_a = var_a * 13!
; Так компилятор ухитрился умножить var_a на 13,
; причем всего за один (!) такт. Также следует отметить, что обе инструкции LEA
; прекрасно спариваются на Pentium MMX и Pentium Pro!

lea         ecx, ds:5[eax*4]
; ECX = EAX*4 + 5

push        edx
push        ecx
; Передаем функции printf var_a * 13 и var_a * 4 +5

shl         eax, 4
; Умножаем var_a на 16

push        eax
push        offset aXXX       ; "%x %x %x\n"
call        _printf
add         esp, 14h
; printf("%x %x %x\n", var_a * 16, var_a * 4 + 5, var_a * 13)

retn

main                endp
```

Этот код, правда, все же не быстрее предыдущего, не оптимизированного, и укладывается в те же три такта, но в других случаях выигрыш может оказаться вполне ощутимым.

Другие компиляторы так же используют LEA для быстрого умножения чисел. Вот, к примеру, Borland поступает так:

```
_main               proc near                   ; DATA XREF: DATA:00407044 o
lea         edx, [eax+eax*2]
; EDX = var_a*3

mov         ecx, eax
; Загружаем в ECX неинициализированную регистровую переменную var_a

shl         ecx, 2
; ECX = var_a * 4

push        ebp
; Сохраняем EBP
```

```
add     ecx, 5
; Добавляем к var_a * 4 значение 5
; К сожалению Borland не использует LEA для сложения.

lea     edx, [eax+edx*4]
; EDX = var_a + (var_a *3) *4 = var_a * 13
; Здесь  Borland и MS единодушны.

mov     ebp, esp
; Открываем кадр стека в середине функции.
; Выше находтся "потерянная" команда push EBP

push    edx
; Передаем printf произведение var_a * 13

shl     eax, 4
; Умножаем ((var_a *4) + 5) на 16
; Здесь происходит глюк компилятора, посчитавшего что: раз переменная var_a
; неинициализирована, то ее можно и не загружать…

push    ecx
push    eax
push    offset aXXX     ; "%x %x %x\n"
call    printf
add     esp, 10h
xor     eax, eax
pop     ebp
retn

_main           endp
```

Хотя "визуально" Borland генерирует не очень “красивый” код, его выполнение укладывается в те же три такта процессора. Другое дело WATCOM, показывающий удручающе отсталый результат на фоне двух предыдущих компиляторов:

```
main            proc near
push    ebx
; Сохраняем EBX в стеке

mov     eax, ebx
; Загружаем в EAX значение неинициализированной регистровой переменной var_a

shl     eax, 2
; EAX = var_a * 4
```

```
sub        eax, ebx
; EAX = var_a * 4 - var_a = var_a * 3
; Здесь WATCOM! Сначала умножает "с запасом", а потом лишнее отнимает!

shl        eax, 2
; EAX = var_a * 3 * 4 = var_a * 12

add        eax, ebx
; EAX = var_a * 12 + var_a = var_a * 13
; Четыре инструкции, в то время как
; Microsoft Visual C++ вполне обходится и двумя!

push       eax
; Передаем printf значение var_a * 13

mov        eax, ebx
; Загружаем в EAX значение неинициализированной регистровой переменной var_a

shl        eax, 2
; EAX = var_a * 4

add        eax, 5
; EAX = var_a * 4 + 5
; Пользоваться LEA WATCOM то же не умеет!

push       eax
; Передаем printf значение var_a * 4 + 5

shl        ebx, 4
; EBX = var_a * 16

push       ebx
; Передаем printf значение var_a * 16

push       offset aXXX       ; "%x %x %x\n"
call       printf_
add        esp, 10h
; printf("%x %x %x\n",var_a * 16, var_a * 4 + 5, var_a*13)

pop        ebx

retn

main_              endp
```

В результате, код, сгенерированный компилятором WATCOM требует шести тактов, т.е. вдвое больше, чем у конкурентов.

1.6 Комплексные операторы

Язык Си\Си++ выгодно отличается от большинства своих конкурентов поддержкой комплексных операторов: x= (где x - любой элементарный оператор), ++ и - -. Комплексные операторы семейства "a x= b" транслируются в "a = a x b" и они идентифицируются так же, как и элементарные операторы. Операторы "++" и "--": в префиксной форме они выражаются в тривиальные конструкции "a = a +1" и "a = a - 1" и не представляющие никакого интереса.

2. Идентификация SWITCH - CASE - BREAK

Для улучшения читабельности программ в язык Си был введен оператор множественного выбора – switch. В Паскале с той же самой задачей справляется оператор CASE.

Легко показать, что switch эквивалентен конструкции "IF (a == x1) THEN оператор1 ELSE IF (a == x2) THEN оператор2 IF (a == x2) THEN оператор2 IF (a == x2) THEN оператор2 ELSE …. оператор по умолчанию". Если изобразить это ветвление в виде логического дерева, то образуется характерная "косичка", прозванная так за сходство с завитой в косу прядью волос – см. рис. 1

Казалось бы, идентифицировать switch никакого труда не составит, – даже не строя дерева, невозможно не обратить внимания на длинную цепочку гнезд, проверяющих истинность условия равенства некоторой переменной с серией непосредственных значений (сравнения переменной с другой переменной switch не допускает).

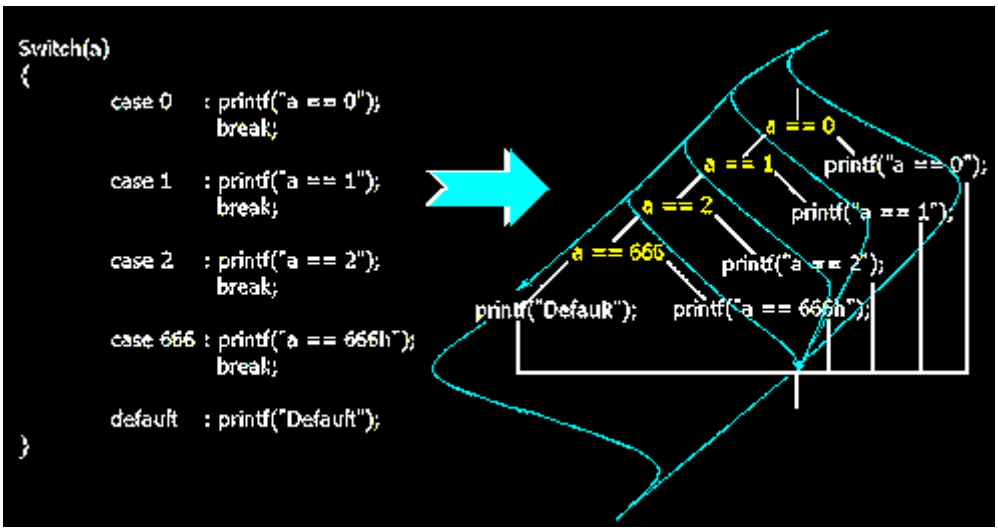


Рисунок 1 Трансляция оператора switch в общем случае

Однако в реальной жизни все происходит совсем не так. Компиляторы (даже не оптимизирующие) транслируют switch в настоящий "мясной рулет", содержащий всевозможные операции отношений. Вот что из этого выйдет, откомпилировав приведенный выше пример компилятором Microsoft Visual C++:

```
main            proc near               ; CODE XREF: start+AF p

var_tmp         = dword ptr -8
var_a           = dword ptr –4

push    ebp
mov     ebp, esp
; Открываем кадр стека

sub     esp, 8
; Резервируем место для локальных переменных
```

```
mov     eax, [ebp+var_a]
; Загружаем в EAX значение переменной var_a

mov     [ebp+var_tmp], eax
; Обратите внимание – switch создает собственную временную переменную!
; Даже если значение сравниваемой переменной в каком-то ответвлении CASE
; будет изменено, это не повлияет на результат выборов!
; В дальнейшем во избежании путаницы, следует условно называть
; переменную var_tmp переменной var_a

cmp     [ebp+var_tmp], 2
; Сравниваем значение переменной var_a с двойкой
; В исходном коде CASE начинался с нуля, а заканчивался 0x666

jg      short loc_401026
; Переход, если var_a > 2
; Обратите на этот момент особое внимание – ведь в исходном тексте такой
; операции отношения не было!
; Причем, этот переход не ведет к вызову функции printf, т.е. этот фрагмент
; кода получен не прямой трансляцией некой ветки case, а как-то иначе!

cmp     [ebp+var_tmp], 2
; Сравниваем значение var_a с двойкой
; Очевидный "прокол" компилятора (обратите внимание никакие флаги не менялись!)

jz      short loc_40104F
; Переход к вызову printf("a == 2"), если var_a == 2
; Этот код явно получен трансляцией ветки CASE 2: printf("a == 2")

cmp     [ebp+var_tmp], 0
; Сравниваем var_a с нулем

jz      short loc_401031
; Переход к вызову printf("a == 0"), если var_a == 0
; Этот код получен трансляцией ветки CASE 0: printf("a == 0")

cmp     [ebp+var_tmp], 1
; Сравниваем var_a с единицей

jz      short loc_401040
; Переход к вызову printf("a == 1"), если var_a == 1
; Этот код получен трансляцией ветки CASE 1: printf("a == 1")

jmp     short loc_40106D
; Переход к вызову printf("Default")
; Этот код получен трансляцией ветки Default: printf("a == 0")
```

```
loc_401026:                             ; CODE XREF: main+10 j
; Эта ветка получает управление, если var_a > 2
cmp     [ebp+var_tmp], 666h
; Сравниваем var_a со значением 0x666

jz      short loc_40105E
; Переход к вызову printf("a == 666h"), если var_a == 0x666
; Этот код получен трансляцией ветки CASE 0x666: printf("a == 666h")

jmp     short loc_40106D
; Переход к вызову printf("Default")
; Этот код получен трансляцией ветки Default: printf("a == 0")

loc_401031:                             ; CODE XREF: main+1C j
; // printf("A == 0")
push    offset aA0          ; "A ==  0"
call    _printf
add     esp, 4
jmp     short loc_40107A
; ^^^^^^^^^^^^^^^^^^^^^^^^ - а вот это оператор break, выносящий управление
; за пределы switch – если бы его не было, то начали бы выполняться все
; остальные ветки CASE, не зависимо от того, к какому значению var_a они
; принадлежат!

loc_401040:                             ; CODE XREF: main+22 j
; // printf("A == 1")
push    offset aA1          ; "A ==  1"
call    _printf
add     esp, 4
jmp     short loc_40107A
; ^ break

loc_40104F:                             ; CODE XREF: main+16 j
; // printf("A == 2")
push    offset aA2          ; "A ==  2"
call    _printf
add     esp, 4
jmp     short loc_40107A
; ^ break

loc_40105E:                             ; CODE XREF: main+2D j
; // printf("A == 666h")
push    offset aA666h       ; "A ==  666h"
call    _printf
add     esp, 4
jmp     short loc_40107A
; ^ break
```

```
loc_40106D:                             ; CODE XREF: main+24 j main+2F j
; // printf("Default")
push    offset aDefault    ; "Default"
call    _printf
add     esp, 4

loc_40107A:                             ; CODE XREF: main+3E j main+4D j ...
; // КОНЕЦ SWITCH
mov     esp, ebp
pop     ebp
; Закрываем кадр стека
retn
main            endp
```

Построив логическое дерево, получаем следующую картину (см. рис. 2). При ее изучении бросается в глаза, во-первых, условие "a >2", которого не было в исходной программе, а во-вторых, изменение порядка обработки case. В то же время, вызовы функций printf следуют один за другим строго согласно их объявлению.

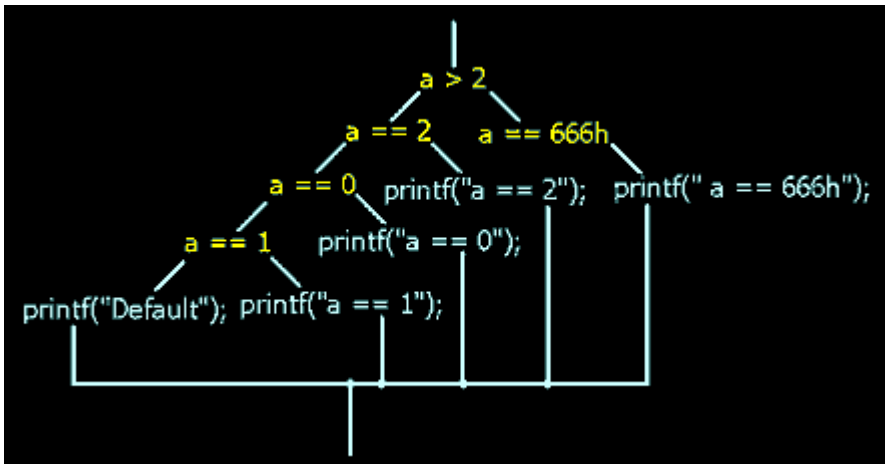


Рисунок 2 Пример трансляция оператора switch компилятором Microsoft Visual C

Назначение гнезда (a > 2) объясняется очень просто – последовательная обработка всех операторов case крайне непроизводительная. Хорошо, если их всего четыре-пять штук, а если программист напишет в switch сотню - другую case? Вот компилятор и "утрамбовывает" дерево, уменьшая его высоту. Вместо одной ветви, изображенной на рис. 2, транслятор построил две, поместив в левую только числа не большие двух, а в правую – все остальные. Благодаря этому, ветвь "666h" из конца дерева была перенесена в его начало. Данный метод оптимизации поиска значений называют "методом вилки".

Изменение порядка сравнений – право компилятора. Стандарт ничего об этот не говорит и каждая реализация вольна поступать так, как ей это заблагорассудится. Другое дело – case-обработчики (т.е. тот код, которому case передает управление в случае истинности отношения). Они обязаны располагаться

так, как были объявлены в программе, т.к. при отсутствии закрывающего оператора break они должны выполняться строго в порядке, замышленном программистом, хотя эта возможность языка Си используется крайне редко.

Таким образом, идентификация оператора switch не сильно усложняется: *если после уничтожения узлового гнезда и прививки правой ветки к левой (или наоборот) получается эквивалентное дерево, и это дерево образует характерную "косичку" – здесь имеет дело оператор множественного выбора или его аналог.*

Весь вопрос в том: правомерно ли удалять гнездо, и не нарушит ли эта операция структуры дерева? Смотрим – на левой ветке узлового гнезда расположены гнезда (a == 2), (a == 0) и (a == 1), а на левом – (a==0x666) Очевидно, если a == 0x666, то a != 0 и a != 1! Следовательно, прививка правой ветки к левой вполне безопасна и после такого преобразования дерево принимает вид типичный для конструкции switch (см. рис. 3)

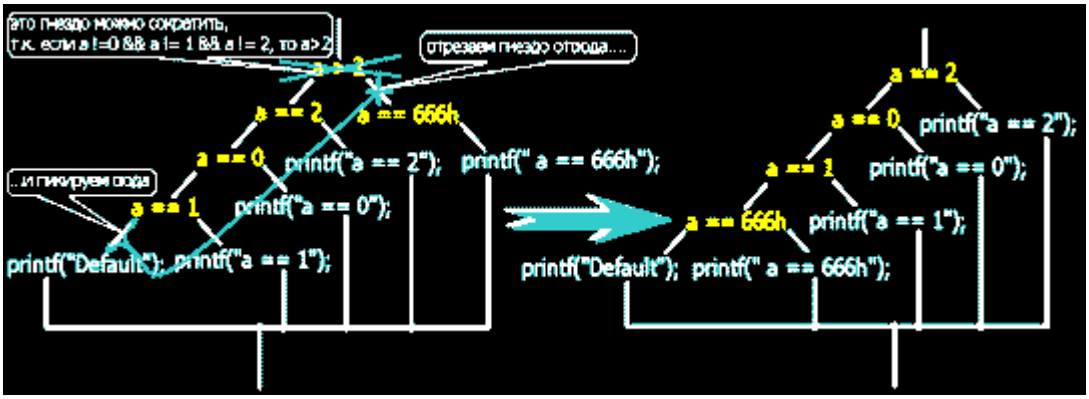


Рисунок 3 Усечение логического дерева

Увы, такой простой прием идентификации срабатывает не всегда! Иные компиляторы могут сделать еще хуже! Если откомпилировать пример компилятором Borland C++ 5.0, то код будет выглядеть так:

```
; int __cdecl main(int argc,const char **argv,const char *envp)
_main                proc near                ; DATA XREF: DATA:00407044 o

push    ebp
mov     ebp, esp
; Открываем кадр стека
; Компилятор помещает нашу переменную a в регистр EAX
; Поскольку она не была инициализирована, то заметить этот факт
; не так-то легко!

sub     eax, 1
; Уменьшает EAX на единицу!
; Никакого вычитания в программе не было!

jb      short loc_401092
; Если EAX < 1, то переход на вызов printf("a == 0")
; (CMP та же команда SUB, только не изменяющая операндов?)
```

```
; Этот код сгенерирован в результате трансляции
; ветки CASE 0: printf("a == 0");
; Внимание, какие значения может принимать EAX, чтобы
; удовлетворять условию этого отношения? На первый взгляд, EAX < 1,
; в частости, 0, -1, -2,… СТОП! Ведь jb - это беззнаковая инструкция
; сравнения! А -0x1 в беззнаковом виде выглядит как 0xFFFFFFFF
; 0xFFFFFFFF много больше единицы, следовательно, единственным подходящим
; значением будет ноль
; Таким образом, данная конструкция - просто завуалированная проверка EAX на
; равенство нулю!

jz      short loc_40109F
; Переход, если установлен флаг нуля
; Он будет установлен в том случае, если EAX == 1
; И действительно переход идет на вызов printf("a == 1")

dec     eax
; Уменьшаем EAX на единицу

jz      short loc_4010AC
; Переход если установлен флаг нуля, а он будет установлен, когда после
; вычитания единицы командой SUB, в EAX останется ровно единица,
; т.е. исходное значение EAX должно быть равно двум
; И верно - управление передается ветке вызова printf("a == 2")!

sub     eax, 664h
; Отнимаем от EAX число 0x664

jz      short loc_4010B9
; Переход, если установлен флаг нуля, т.е. после двукратного уменьшения EAX
; равен 0x664, следовательно, исходное значение - 0x666

jmp     short loc_4010C6
; прыгаем на вызов printf("Default"). Значит, это - конец switch

loc_401092:                             ; CODE XREF: _main+6 j
; // printf("a==0");
push    offset aA0          ; "a == 0"
call    _printf
pop     ecx
jmp     short loc_4010D1

loc_40109F:                             ; CODE XREF: _main+8 j
; // printf("a==1");
push    offset aA1          ; "a == 1"
call    _printf
pop     ecx
jmp     short loc_4010D1
```

```
loc_4010AC:                             ; CODE XREF: _main+B j
; // printf("a==2");
push    offset aA2          ; "a == 2"
call    _printf
pop     ecx
jmp     short loc_4010D1

loc_4010B9:                             ; CODE XREF: _main+12 j
; // printf("a==666");
push    offset aA666h       ; "a == 666h"
call    _printf
pop     ecx
jmp     short loc_4010D1

loc_4010C6:                             ; CODE XREF: _main+14 j
; // printf("Default");
push    offset aDefault     ; "Default"
call    _printf
pop     ecx

loc_4010D1:                             ; CODE XREF: _main+21 j          _main+2E
j ...
xor     eax, eax
pop     ebp
retn
_main               endp
```

Код, сгенерированный компилятором, модифицирует сравниваемую переменную в процессе сравнения! Оптимизатор посчитал, что DEC EAX короче, чем сравнение с константой, да и работает быстрее. Прямая ретрансляция кода дает конструкцию вроде: "if (a-- == 0) printf("a == 0"); else if (a==0) printf("a == 1"); else if (--a == 0) printf("a == 2"); else if ((a-=0x664)==0) printf("a == 666h); else printf("Default")", - в которой совсем не угадывается оператор switch! Впрочем, угадать его возможно. Где есть длинная цепочка "IF-THEN-ELSE-IF-THEN-ELSE…". Узнать оператор множественного выбора будет еще легче, если изобразить его в виде дерева (см. рис. 4) , характерная "косичка"!

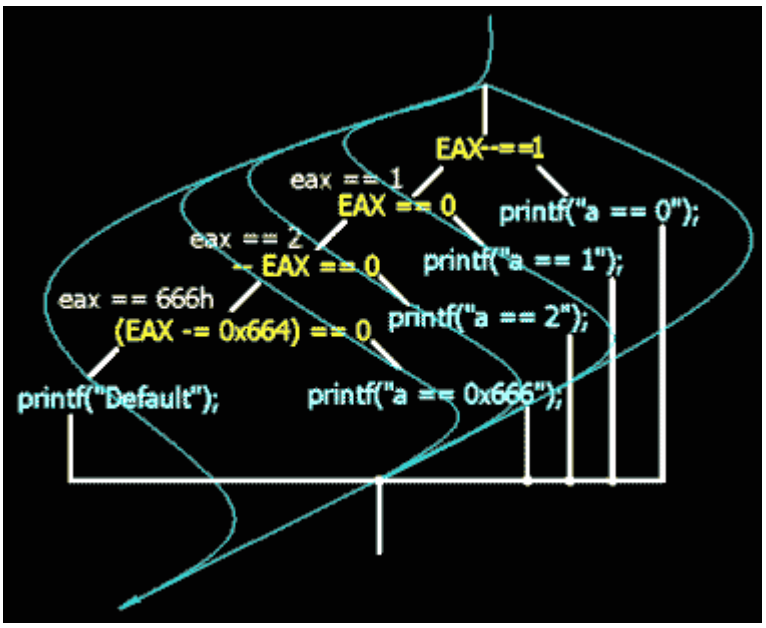


Рисунок 4 Построение логического дерева с гнездами, модифицирующими саму сравниваемую переменную

Другая характерная деталь - case-обработчики, точнее оператор break традиционно замыкающий каждый из них. Они-то и образуют правую половину "косички", сходясь все вместе с точке "Z". Правда, многие программисты питают паралогическую любовь к case-обработчикам размером в два-три экрана, включая в них помимо всего прочего и циклы, и ветвления, и даже вложенные операторы множественно выбора! В результате правая часть "косички" превращается в непроходимый таежный лес. Но даже если и так - левая часть "косички", все равно останется достаточно простой и легко распознаваемой!

В заключение темы рассмотрим последний компилятор - WATCOM C. Как и следует ожидать, здесь подстерегают свои тонкости. Итак, откомпилированный им код предыдущего примера должен выглядеть так:

```
main_           proc near               ; CODE XREF: __CMain+40 p
push    8
call    __CHK
; Проверка стека на переполнение

cmp     eax, 1
; Сравнение регистровой переменной EAX, содержащей в себе переменную a
; со значением 1

jb      short loc_41002F
; Если EAX == 0, то переход к ветви с дополнительными проверками

jbe     short loc_41003A
; Если EAX == 1 (т.е. условие bellow уже обработано выше), то переход
; к ветке вызова printf("a == 1");

cmp     eax, 2
```

```
; Сравнение EAX со значением 2

jbe         short loc_410041
; Если EAX == 2 (условие EAX <2 уже было обработано выше),  то переход
; к ветке вызова printf("a == 2");

cmp       eax, 666h
; Сравнение EAX со значением 0x666

jz          short loc_410048
; Если EAX == 0x666, то переход к ветке вызова printf("a == 666h");

jmp       short loc_41004F
; Ни одно из условий не подошло - переход к ветке "Default"


loc_41002F:                                ; CODE XREF: main_+D j
; // printf("a == 0");
test        eax, eax
jnz         short loc_41004F
; Совершенно непонятно - зачем здесь дополнительная проверка?!
; Это ляп компилятора - она ни к чему!

push      offset aA0          ; "A ==  0"
; Здесь WATCOM сумел обойтись всего одним вызовом printf!
; Обработчики case всего лишь передают ей нужный аргумент!
; Вот это действительно - оптимизация!
jmp       short loc_410054

loc_41003A:                                ; CODE XREF: main_+F j
; // printf("a == 1");
push      offset aA1          ; "A ==  1"
jmp       short loc_410054

loc_410041:                                ; CODE XREF: main_+14 j
; // printf("a == 2");
push      offset aA2          ; "A ==  2"
jmp       short loc_410054

loc_410048:                                ; CODE XREF: main_+1B j
; // printf("a == 666h");
push      offset aA666h     ; "A ==  666h"
jmp       short loc_410054

loc_41004F:                                ; CODE XREF: main_+1D j          main_+21
j
; // printf("Default");
push      offset aDefault    ; "Default"
```

```
loc_410054:                             ; CODE XREF: main_+28 j         main_+2F
j ...
call    printf_
; Это printf, получающий аргументы из case-обработчиков!

add     esp, 4
; Закрытие кадра стека

retn
main_           endp
```

В общем, WATCOM генерирует более хитрый, но, как ни странно, весьма наглядный и читабельный код.

2.1 Отличия switch от оператора case языка Pascal

Оператор CASE языка Pascal практически идентичен своему Си собрату - оператору switch, хотя и близнецами их не назовешь: оператор CASE выгодно отличается поддержкой *наборов и диапазонов значений.* Ну, если обработку наборов можно реализовать и посредством switch, правда не так элегантно как на Pascal, то проверка вхождения значения в диапазон на Си организуется исключительно с помощью конструкции "IF-THEN-ELSE". Зато в Паскале каждый case-обработчик принудительно завершается неявным break, а Си-программист волен ставить (или не ставить) его по своему усмотрению.

```
CASE a OF                               switch(a)
begin                                   {
1       : WriteLn('a == 1');            case 1    : printf("a == 1");
  break;
2,4,7   : WriteLn('a == 2|4|7');        case 2    :
case 4  :
case 7  : printf("a == 2|4|7");
  break;
9       : WriteLn('a == 9');            case 9    : printf("a == 9");
  break;
end;
```

Однако оба языка накладывают жесткое ограничение на выбор сравниваемой переменной: она должна принадлежать к перечисленному типу, а все наборы (диапазоны) значений представлять собой константы или константные выражения, вычисляемые на стадии компиляции. Подстановка переменных или вызовов функций не допускается.

Представляет интерес посмотреть: как Pascal транслирует проверку диапазонов и сравнить его с компиляторами Си. Рассмотрим следующий пример:

```
VAR
```

```
a : LongInt;
BEGIN

CASE a OF
2               :       WriteLn('a == 2');
4, 6            :       WriteLn('a == 4 | 6 ');
10..100         :       WriteLn('a == [10,100]');
END;
END.
```

Результат его компиляции компилятором Free Pascal должен выглядеть так (приведена лишь левая часть "косички"):

```
mov     eax, ds:_A
; Загружаем в EAX значение сравниваемой переменной

cmp     eax, 2
; Сравниваем EAX со значением 0x2

jl      loc_CA          ; Конец CASE
; Если EAX < 2, то - конец CASE

sub     eax, 2
; Вычитаем из EAX значение 0x2

jz      loc_9E          ; WriteLn('a == 2');
; Переход на вызов WriteLn('a == 2') если EAX == 2

sub     eax, 2
; Вычитаем из EAX значение 0x2

jz      short loc_72    ; WriteLn('a == 4 | 6');
; Переход на вызов WriteLn("a == 4 | 6') если EAX == 2 (соотв. a == 4)

sub     eax, 2
; Вычитаем из EAX значение 0x2

jz      short loc_72    ; WriteLn('a == 4 | 6');
; Переход на вызов WriteLn("a == 4 | 6') если EAX == 2 (соотв. a == 6)

sub     eax, 4
; Вычитаем из EAX значение 0x4

jl      loc_CA          ; Конец CASE
; Переход на конец CASE, если EAX < 4 (соотв. a < 10)

sub     eax, 90
; Вычитаем из EAX значение 90
```

```
jle         short loc_46        ; WriteLn('a = [10..100]');
; Переход на вызов WriteLn('a = [10..100]') если EAX <= 90 (соотв. a <= 100)
; Поскольку, случай a > 10 уже был обработан выше, то данная ветка
; срабатывает при условии a>=10 && a<=100.

jmp         loc_CA              ; Конец CASE
; Прыжок на конец CASE - ни одно из условий не подошло
```

Как видно, Free Pascal генерирует практически тот же самый код, что и компилятор Borland C++ 5.x, поэтому его анализ не должен вызвать никаких сложностей.

2.2 Обрезка (балансировка) длинных деревьев

В некоторых (хотя и редких) случаях, операторы множественного выбора содержат сотни (а то и тысячи) наборов значений, и если решать задачу сравнения "в лоб", то высота логического дерева окажется гигантской до неприличия, а его прохождение займет весьма длительное время, что не лучшим образом скажется на производительности программы.

Здесь возникает вопрос: чем собственно занимается оператор switch? Если отвлечься от устоявшейся идиомы "*оператор SWITCH дает специальный способ выбора одного из многих вариантов, который заключается в проверке совпадения значения данного выражения с одной из заданных констант и соответствующем ветвлении"*, то можно сказать, что switch - оператор *поиска* соответствующего значения. В таком случае каноническое switch - дерево представляет собой тривиальный алгоритм последовательного поиска - самый неэффективный алгоритм из всех.

Пусть, например, исходный текст программы выглядел так:

```
.
switch (a)
{
case 98 : …;
case 4  : …;
case 3  : …;
case 9  : …;
case 22 : …;
case 0  : …;
case 11 : …;
case 666: …;
case 096: …;
case 777: …;
case 7  : …;
}
```

Тогда соответствующее ему не оптимизированное логическое дерево будет достигать в высоту одиннадцати гнезд (см. рис. 5 слева). Причем, на левой ветке корневого гнезда окажется аж десять других гнезд, а на правой - вообще ни одного (только соответствующий ему case - обработчик).

Исправить "перекос" можно разрезав одну ветку на две и привив образовавшиеся половинки к новому гнезду, содержащему условие, определяющее в какой из веток следует искать сравниваемую переменную. Например, левая ветка может содержать гнезда с четными значениями, а правая - с нечетными. Но это плохой критерий: четных и нечетных значений редко бывает поровну и вновь образуется перекос. Гораздо надежнее поступить так: берется наименьшее из всех значений и бросается в кучу *A*, затем берется наибольшее из всех значений и бросается в кучу *B*. Так повторяется до тех пор, пока не рассортируются все, имеющиеся значения.

Поскольку оператор множественного выбора требует уникальности каждого значения, т.е. каждое число может встречаться в наборе (диапазоне) значений лишь однажды, легко показать, что:

а) в обеих кучах будет содержаться равное количество чисел (в худшем случае - в одной куче окажется на число больше);

б) все числа кучи *A* меньше наименьшего из чисел кучи *B*. Следовательно, достаточно выполнить только одно сравнение, чтобы определить в какой из двух куч следует искать сравниваемое значения.

Высота нового дерева будет равна ((N+1/2)+1), где N - количество гнезд старого дерева. Действительно, ветвь дерева делится надвое и добавляется новое гнездо - отсюда и берется N/2 и +1, а (N+1) необходимо для округления результата деления в большую сторону. Т.е. если высота не оптимизированного дерева достигала 100 гнезд, то теперь она уменьшилась до 51. Затем можно разбить каждую из двух ветвей еще на две. Это уменьшит высоту дерева до 27 гнезд! Аналогично, последующее уплотнение даст 16 ' 12 ' 11 ' 9 ' 8… и все! Более плотная упаковка дерева невозможна. Восемь гнезд - это не сто! Полное прохождение оптимизированного дерева потребует менее девяти сравнений!

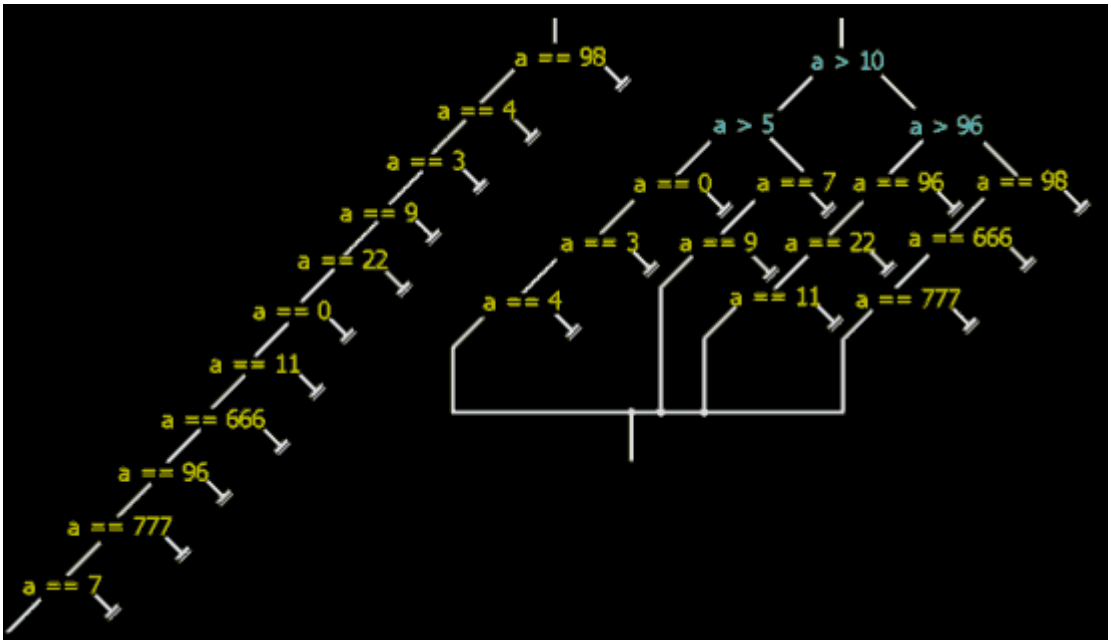
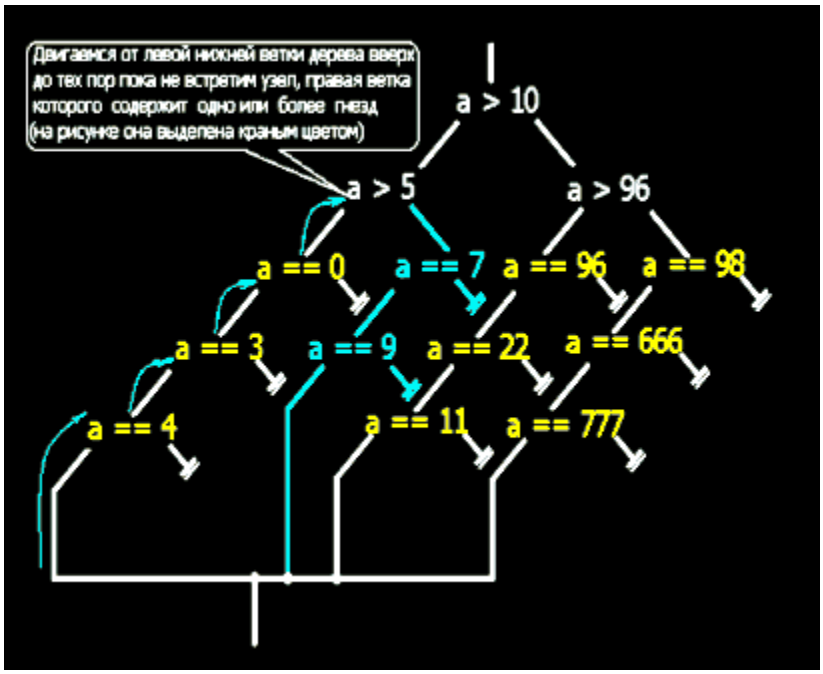


Рисунок 5 Логическое дерево до утрамбовки (слева) и после (справа)

"Трамбовать" логические деревья оператора множественного выбора умеют практически все компиляторы - даже не оптимизирующие! Это увеличивает

производительность, но затрудняет анализ откомпилированной программы. Взглянув еще раз на рис. 5 - левое несбалансированное дерево наглядно и интуитивно - понятно. После же балансировки (правое дерево) совсем запутанное.
К счастью, балансировка дерева допускает эффективное обращение. Но прежде, чем разбирать деревья введем понятие *балансировочного узла*. Балансировочный узел не изменяет логики работы двоичного дерева и являются факультативным узлом, единственная функция которого укорачивание длины ветвей. Балансировочный узел без потери функциональности дерева может быть замещен любой из своих ветвей. Причем каждая ветвь балансировочного узла должна содержать одно или более гнезд.

Рассуждая от противного - все узлы логического дерева, правая ветка которых содержит одно или более гнезд, могут быть замещены на эту самую правую ветку без потери функциональности дерева, то данная конструкция представляет собой оператор switch. Правая ветка потому что оператор множественного выбора в "развернутом" состоянии представляет цепочку гнезд, соединенных левыми ветвями друг с другом, а на правых держащих case-обработчики, - вот и следует попытаться подцепить все правые гнезда на левую ветвь. Если это удается, значит, имеет место оператор множественного выбора, если нет – то с чем-то другим.
Рассмотрим обращение балансировки на примере следующего дерева (см. рис. 6-а). Двигаясь от левой нижней ветви, следует продолжать взбираться на дерево до тех пор, пока не встретится узел, держащий на своей правой ветви одно или более гнезд. В данном случае - это узел (a > 5). Если данный узел заменить его гнездами (a==7) и (a == 9) функциональность дерева не нарушиться! (см. рис. 6-б). Аналогично узел (a > 10) может быть безболезненно заменен гнездами (a > 96), (a == 96), (a == 22) и (a == 11), а узел (a > 96) в свою очередь - гнездами (a == 98), (a == 666) и (a == 777). В конце -концов образуется классическое switch-дерево, в котором оператор множественного выбора распознается с первого взгляда.



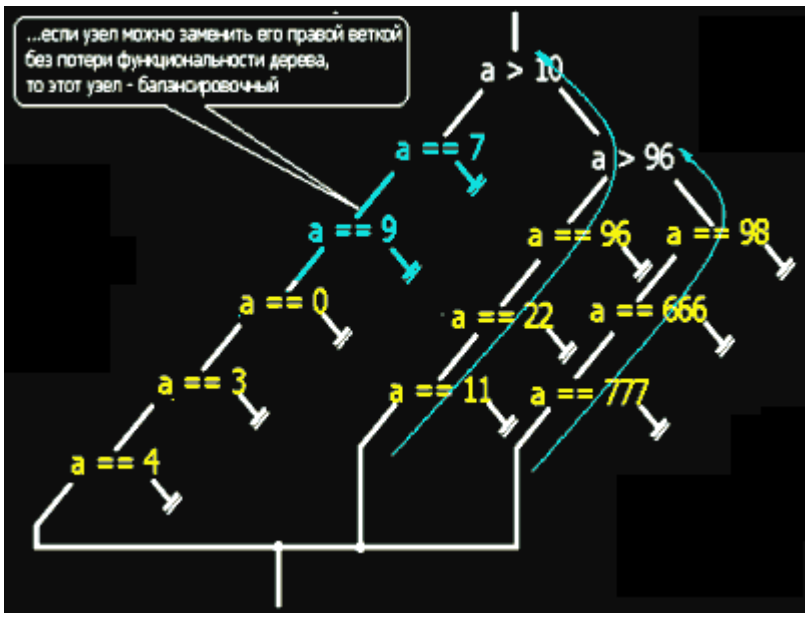


Рисунок 6-а Обращение балансировки логического дерева

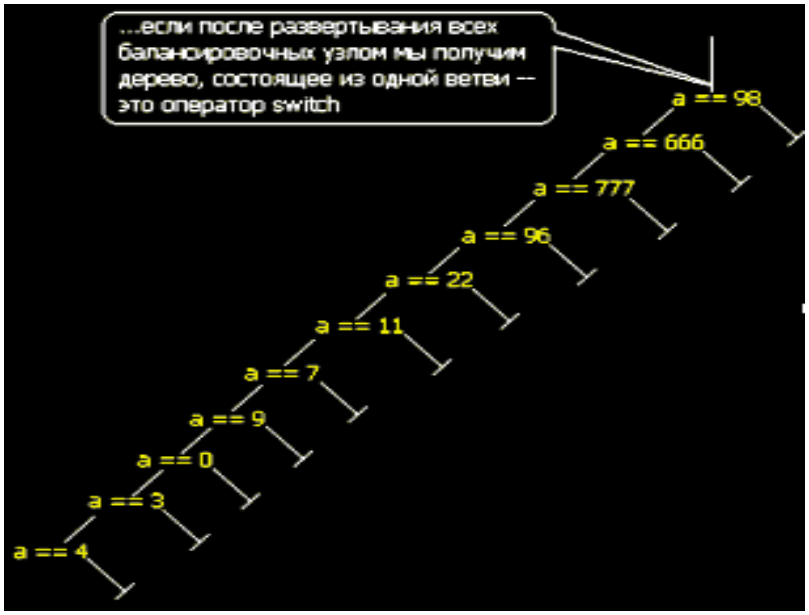


Рисунок 6-б Обращение балансировки логического дерева

2.3 Сложные случаи балансировки или оптимизирующая балансировка

Для уменьшения высоты "утрамбовываемого" дерева хитрые трансляторы стремятся замещать уже существующие гнезда балансировочными узлами. Рассмотрим следующий пример: (см. рис. 7). Для уменьшения высоты дерева транслятор разбивает его на две половины - в левую идут гнезда со значениями меньшие или равные единицы, а в правую - все остальные. Казалось бы, на правой ветке узла (a > 1) должно висеть гнездо (a == 2), но это не так! Здесь видно узел (a >2), к левой ветки которого прицеплен case-обработчик :2. Вполне логично - если (a > 1) и !(a > 2), то a == 2!

Легко видеть, что узел (a > 2) жестко связан с узлом (a > 1) и работает на пару с последним. Нельзя выкинуть один из них, не нарушив работоспособности

другого! Обратить балансировку дерева по описанному выше алгоритму без нарушения его функциональности невозможно! Отсюда может создаться мнение, что имеет место вовсе не оператор множественного выбора, а что-то другое.
Чтобы развеять это заблуждение придется предпринять ряд дополнительных шагов. Первое - у switch-дерева все case-обработчики всегда находятся на правой ветви. Нужно посмотреть - можно ли трансформировать дерево так, чтобы case-обработчик 2 оказался на левой ветви балансировочного узла? Оказывается, можно: заменив (a > 2) на (a < 3) и поменяв ветви местами (другими словами выполнив инверсию). Второе - все гнезда switch-дерева содержат в себе условия равенства, - смотрим: можно ли заменить неравенство (a < 3) на аналогичное ему равенство? Да, можно - (a == 2)!

Вот, после всех этих преобразований, обращение балансировки дерева удается выполнить без труда!

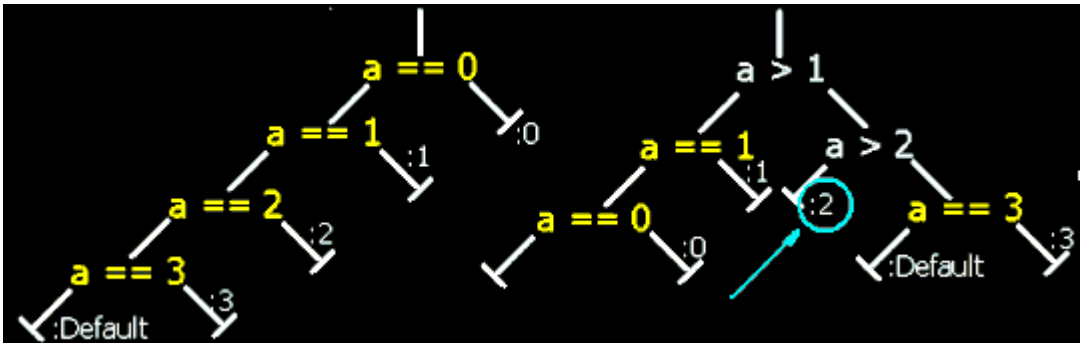


Рисунок 7 Хитрый случай балансировки

2.4 Ветвления в case-обработчиках.

В реальной жизни case-обработчики прямо-таки кишат ветвлениями, циклами и прочими условными переходами всех мастей. Как следствие - логическое дерево приобретает вид ничуть не напоминающий оператор множественного выбора. Идентифицировав case-обработчики, можно бы решить эту проблему, но их нужно идентифицировать!

За редкими исключениями, case-обработчики не содержат ветвлений относительно сравниваемой переменной. Действительно, конструкции "switch(a) …. case 666 : if (a == 666) …." или "switch(a) …. case 666 : if (a > 66) …." абсолютно лишены смысла. Таким образом, можно смело удалить из логического дерева все гнезда с условиями, не касающимися сравниваемой переменной (переменной коневого гнезда).

Если программист в порыве собственной глупости или стремлении затруднит анализ программы "впаяет" в case-обработчики ветвления относительно сравниваемой переменной, то оказывается, это ничуть не затруднит анализ! "Впаянные" ветвления элементарно распознаются и обрезаются либо как избыточные, либо как никогда не выполняющиеся. Например, если к правой ветке гнезда (a == 3) прицепить гнездо (a > 0) - его можно удалить, как не несущее в себе никакой информации. Если же к правой ветке того же самого гнезда прицепить гнездо (a == 2) его можно удалить, как никогда не выполняющееся - если a == 3, то заведомо a != 2!

Заключение

Процесс дизассемблирования является важной частью при программировании и компиляции исходных текстов программ, особенно для начинающих программистов. Он позволяет получить исходные коды практически любого откомпилированного приложения и сделать подробный его анализ.

Так же дизассемблирование позволяет выбрать наиболее эффективный продукт для компиляции и сборки приложений с целью оптимизации и быстродействия конечного программного продукта.

В работе был произведен подробный анализ дизассемблированного кода с целью идентификации математических операций и операторов SWITCH - CASE – BREAK. Были рассмотрены все варианты сборки программ разными трансляторами, и установлено что наилучшим средством компилирования приложений является компилятор от фирмы Microsoft.

Список используемой литературы

Практическая работа №5

Использование отладчика Win32Dasm и редактора HxD для обхода ввода серийного номера

Цель работы: преобразовать исполняемый модуль program.exe таким образом, чтобы обойти проблемы с введением серийного номера и использовать программу.

Задачи: Найти в program.exe команды, отвечающие за проверку серийного номера и исправить их таким образом, чтобы эта проверка не осуществлялась.

Инструменты: Отладчик Win32Dasm – дизассемблирует исполняемый модуль и позволяет его выполнять в пошаговом режиме. HxD – редактор файлов.

Шаги выполнения работы:

1. Открыть отладчик Win32Dasm и загрузить исполняемый модуль program.exe (“Дизассемблер” – “Открыть файл…”).
2. Выполнить пошаговое выполнение программы до вывода окна с вводом серийного номера, ввести серийный номер (любое значение) и далее проследить какие операции условного перехода будут после этого (“Отладка” – “Загрузить процесс” – “Чтение” и далее F8 – шаг с заходом в процедуры или F7 – шаг без захода в процедуры). Первая колонка показывает смещение команды в памяти, вторая – команда в двоичном виде, третья – команда на языке ассемблера.
3. Открыть в редакторе HxD исполняемый модуль program.exe. Найти команду условного перехода и заменить ее на команду безусловного перехода. Сохранить модуль. (Внимание, следует отметить, что первая колонка показывает смещение относительно начала файла, а не относительно начала размещения в памяти как в Win32Dasm).
4. Запустить program.exe, убедиться в том, что она работает.

Некоторые двоичные коды команд условного и безусловного перехода

je _metka	04 74h	условный переход
jne _metka	02 75h	условный переход
jmp _metka	00 EBh	безусловный переход

_metka