ReVision Goals

Olaf Bernstein

June 3, 2018

The Goal of this project is the complete revision of my favourite concepts of programming languages. It will be primarily oriented around C++ concepts because its currently one of my favourite and most used programming languages. I'll also take some inspiration from other languages like GO.

Contents

1 Why Not C++? 2

2 Control Flow 2
2.1 *while* 2
2.2 *jmp* 2
2.3 *for* 3
2.4 *switch* 3
2.5 *if* 3
2.6 *do* 3
2.7 *asm* 4

3 Type Modifier 4
3.1 *const* 4
3.2 '*[]*' 4
3.3 '*[]!*' 5
3.4 '***' 5
3.5 '**!*' 5
3.6 Map 5

4 Primitive Types 5
4.1 Integer 5
4.2 Floating Point 6
4.3 Character/String Types 6

5 Abstract Types 6
5.1 Composite Types 6
5.1.1 *class* 6
5.1.2 *struct* 6
5.1.3 *enum* 6
5.1.4 *union* 6
5.2 String 6

6 Operators 6
6.1 *new* 6
6.2 *delete* 6

7 Scope 6
7.1 *namespace* 6
7.2 *use* 6

8 Other 7
8.1 *auto* 7

1 Why Not C++?

So why not C++, well C++ might support a grate amount of features, heck you can even make thinks like an NES emulator at compile time, but it has gotten pretty messy lately. For Example, there are still trying to keep the C backwards compatibility, but have extremely many new features. Some serve the same purpose of the C equivalent, but are only there to support the new C++ features.

2 Control Flow

2.1 *while*

1. If condition is false jmp 4.
2. Execute code block.
3. jmp 1.
4. Continue execution.

Syntax:

```
while(condition)
{
        // Some code ...
}
```

2.2 *jmp*

Will jump to the Lable that's specified.

```
LABLE:
// Some code ...
jmp LABLE;
```

With the *jmpif* keyword can also insert a optional condition.

```
LABLE:
// Some code ...
jmpif(condition) LABLE;
```

2.3 *for*

1. Execute initialization code.
2. If condition is false jmp 5.
3. Execute code.
4. Execute iterate code.
5. Continue execution.

```
for(initialization; condition; iterate)
{
        // Some code ...
}
```

2.4 *switch*

Generates a jump table that jumps to the cases where the value is the same as the variable value.

```
switch(variable)
{
        case value_1:
        {
                // Some code ...
        }
        case value_2, value_3:
        // Some code ...

        default:
        // Some code ...
}
```

2.5 *if*

1. If the condition is true execute next code block.
2. Optional else code block get executed if 1 is false.

Is also possible to stack if statements using else if's.

```
if(condition_1)
{
        // Some code ...
}
else if(condition_2)
{
        // Some code ...
}
else
{
        // Some code ...
}
```

2.6 *do*

1. Execute code block.
2. If condition is true jmp 1.
4. Continue execution.
Syntax:

```
do
{
        // Some code ...
} while(condition);
```

2.7 *asm*

The code block after the asm keyword will be executed as assembly code.

```
asm
{
        mov eax, 2 ; ...
};
```

3 Type Modifier

3.1 *const*

Can be used to make a type imitable. On pointers it makes the pointer unable to point to a different memory position.

```
const float pi = 3.14;
// float = 2;  Error this would be invalid syntax.

int a = 10;
int b = 21;
const * int ptr = &a;
// ptr = &b; Error this would be invalid syntax.
```

3.2 *'[]'*

Indicates that the type is an array. There are many different syntax styles:

```
[10]int array0; // creates an array of size 10 with the type of \textit{int).

[10]int array1 = {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
};
[]int array2 = { // The size of an array can be automaticly deduced.
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
};
// array1 and array2 have exactly the same value and size.

[256]int array3 = {
        [4] = 2, // Will set the 4th index to the value 2.
        [2,6] = 4, // Works with n indexes at one time.
        5 // Will set the next index in this case the 7th to 5.
};


[2][2]int array2d0 = { // You can also create array of arrays. And so on...
        {1, 2},
        {3, 4}
};
[][]int array2d1 = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
};
[][]int array2d2 = {
        [2] = {[1] = 2, [3,7] = 1},
        [4] = 7
};
```

3.3 '*[]!*'

Vector

3.4 '***'

Pointer

3.5 '**!*'

Unique Pointer

3.6 Map

[](int—char)

4 Primitive Types

4.1 Integer

signed	unsigned	Description
byte	*ubyte*	
int8	*uint8*	8-Bit integer value.
short	*ushort*	
int16	*uint16*	16-Bit integer value.
int	*uint*	
int32	*uint32*	32-Bit integer value.
long	*ulong*	
int64	*uint64*	64-Bit integer value if 64 bits are available otherwise 32 bits. Using this data type on an system which doesn't support 64 bits will also grant an compiler Warning.

4.2 Floating Point

Type	Description
float	IEEE-32 bit Floating Point.
double	IEEE-64 bit Floating Point.

4.3 Character/String Types

Type	Description
utf8	
char	8-Bit unicode character.
utf16	16-Bit unicode character.
utf32	32-Bit unicode character.
wchar	Depending on implementation same as utf16 or utf32 type.

5 Abstract Types

5.1 Composite Types

5.1.1 *class*

5.1.2 *struct*

5.1.3 *enum*

5.1.4 *union*

5.2 String

Type	Description
u8string	
string	Uses *char* for each character.
wstring	Uses *wchar* for each character.

6 Operators

6.1 *new*

6.2 *delete*

7 Scope

7.1 *namespace*

7.2 *use*

The scope of the code block gets reduced to the variables in the capture list.

```
use x, y // Capture list
{
        // Some code ...
};
```

Optional you can return a variable just like in a function.

```
z = use x, y // Capture list
{
        // Some code ...
        return val;
};
```

8 Other

8.1 *auto*