RPP

1.1+2=3
2.C
3.面向对象
4.Lisp
5.省略括号
6.常量对象
7.运算符 DIY
8.无缝内联汇编
9.伪代码
10.模板函数
11.宏
12.函数指针
13.动态调用函数
14.元函数
15.JS
16.反射
17.闭包
18.多重继承
19.变参函数
20.默认参数
21.多线程
22.本地调用
23.类型转换
24.动态数组
25.多参数组
26.重载
27.指针和引用
28.全局变量
29.成员变量
30.局部变量
31.常量
32.控制结构
33.模板类
34.包
35.堆
36.库
37.编译和链接
38.裸奔
39.图形界面
40.文言文
41.语言特性
42.函数对象
43.效率
44.惰性求值

1. 1+2=3

第一个 RPP 程序：

```
void main()
{
   a=1
   b=2
   puts(a+b)
}
```

当然也可以使用 int main()。

2. C

RPP 支持 C 风格，上一个程序可以写成这样：

```
void main()
{
   int a=1;
   int b=2;
   puts(a+b);
}
```

对比上一节的程序可以看到 RPP 的一些特点：
*更彻底地类型推断，C++11 需要使用关键字 auto，RPP 可以省略 auto
*函数定义和函数调用均可省略后面的小括号
*语句后面的分号也可以省略
*对于返回值为 void 的函数，可以省略 void（标准 C 默认返回 int）

3. 面向对象

RPP 支持 C#（Java）风格，上一个程序可以写成这样：

```
public class main
{
   public static void main()
   {
      int a=1;
      int b=2;
      puts(a+b);
   }
}
```

4. Lisp

RPP 内部使用 Lisp 作为中间层，因此上一个程序可以写成这样：

```
void main()
{
   int a
   int b
   [int = [a 1]]
   [int = [b 2]]
   [rf print [[int + [a b]]]]
}
```

其中

```
[rf print [[int + [a b]]]]
```

等价于

```
rf.print(a+b)
```

另一个 S 表达式的例子:

```
example\4_2.h
```

5. 省略括号

控制结构（if/for/while）后面的小括号是可以省略的，比如有一个递归求和的函数：

```
int sum(int a)
{
   if a<=1
      return 1
   return sum(a-1)+a
}
```

那么

```
if a<=1
```

等价于

```
if(a<=1)
```

RPP 支持各种花括号风格，也支持类似 python 的控制结构省略花括号，这时编译器将使用缩进区分语句块，缩进可以用 tab 或者空格表示，1 个 tab 等于 4 个空格，比如：

```
if 1
   puts 2
   puts 3
```

等价于

```
if 1 {
   puts 2
   puts 3
}
```

等价于

```
if(1)
{
   puts(2);
   puts(3);
}
```

如果函数调用的右边没有小括号，则本行后面所有单词均作为函数的参数，比如：

```
puts(sum(9))
```

等价于

```
puts sum(9)
```

等价于

```
puts sum 9
```

表达式过长折行的时候不能省略括号（左括号不可折行）：

```
if(1||
   2)
   puts "true"

func(1,
   2)

func(1,2
   )
```

6. 常量对象

常量可以作为对象使用，这只是编译器提供的一个语法糖，比如

```
2.print
```

会替换为

```
int(2).print
```

而

```
"123"+"abc"
```

会替换为

```
rstr("123")+rstr("abc")
```

和 C++一样，类名后面接小括号表示生成临时对象，下面是生成一个临时的 rstr 对象：

```
rstr("123")
```

生成一个临时的 int 对象：

```
int(2)
```

7. 运算符 DIY

RPP 支持自定义新的运算符，也可以自定义运算符的优先级，请手动修改 rinf\optr.txt 这个文件。比如：

```
%
2
```

这个%后面跟的数字就是求余运算符的优先级，修改这个数字即可。但不要调换里面运算符的顺序，因为有些运算符是编译器使用的。如果要增加新的运算符，请在文件末尾增加两行。（不能自定义>>运算符）

借助于 RPP 的运算符自定义功能可以实现一些很有趣的功能，比如：

```
D=(A∩B)∪(A∩C)
```

这是数学书上的一个表达式，在 RPP 里只要自定义∪和∩两个运算符然后设定好优先级，再写两个运算符函数：

```
friend vector<T> operator∪(vector<T>& a,vector<T>& b)
{
    ...
}

friend vector<T> operator∩(vector<T>& a,vector<T>& b)
{
    ...
}
```

这段表达式就是可以运行的。看起来就像是你自己定义了一个专用于数学集合运算的新语言，因此，RPP 这种灵活的语法很适合实现领域特定语言（DSL）。

RPP 相同优先级的运算符都是从左往右运算的，C 语言可以这样：

```
**p=2
```

但 RPP 只能这样：

```
*(*p)=2
```

注意 RPP 的乘法运算符和指针运算符都是*，优先级相同，均为 2。而标准 C 乘法运算符优先级是 2，指针运算符优先级是 1，两者明显有区别。

RPP 的系统库也不支持连续赋值，C 语言里可以这样：

```
a=b=2
```

RPP 只能这样：

```
b=2
a=b
```

如果一定要连续赋值，可以修改 rsrc\int.h 这个文件，将 operator=这个函数改成这样：

```
int& operator=(int a)
{
   mov esi,this
   mov [esi],a
   return this
}
```

然后就可以连续赋值（注意括号是必须的）：

```
a=(b=2)
```

可以看到这样的话多返回了一个引用，影响效率。

rsrc\int.h 里面还添加一个反向赋值的运算符函数：

```
friend int& operator=>(int a,int& this)
{
   mov esi,this
   mov [esi],a
   return this
}
```

那么使用这个运算符就可以连续反向赋值：

```
2=>a=>b
```

这样 a 和 b 都会赋值为 2，其实从左往右赋值有时看起来更直观一些。

打开 rsrc\basic.h 可以找到：

```
define <- =
define <> !=
```

可见赋值运算符还有另一种写法：

```
a <- 2
```

其等价于

```
a = 2
```

而不等于运算符也有另一种写法：

```
a != 2
```

等价于

```
a <> 2
```

BASIC 语言的<>好像看起来更直观一些。

标准 C 中减法运算符和负号运算符都是 - ，而 RPP 中减法运算符是 - ，负号运算符是 neg ， 下面 3 句是等价的：

```
a=-2
a=(-2)
a=neg 2
```

对于加了括号的常量表达式，会在编译阶段进行求值。

8. 无缝内联汇编

在上一节的例子中可以看到，RPP 用 this 引用代替了 C++的 this 指针，实际上 RPP 内部引用和指针都是 4 个字节的地址（32 位），只是外部访问的时候有些区别。C++是这样：

```
this->print
```

RPP 则是：

```
this.print
```

RPP 支持无缝内联汇编，C 内联汇编需要用关键字 asm：

```
asm mov esi,a
```

或者

```
asm
{
   mov esi,a
}
```

RPP 可以省略关键字 asm：

```
mov esi,a
```

a 是一个局部变量，编译器会把它替换为：

```
mov esi,[ebp+n]
```

其中 n 是局部变量 a 的偏移。

RPP 的栈空间安排是这样的：（地址从低地址到高地址）

ebp
局部变量 1
局部变量 2
……
局部变量 n
返回地址
函数参数 1
函数参数 2
……
函数参数 n
返回值

返回值由调用者析构，而函数参数和局部变量由被调用者析构，函数参数从右往左入栈。注意和 stdcall 以及 cdecl 都是有区别的，RPP 的返回值统一放在堆栈中传递，而不是通过 eax。

举个例子：

```
friend int operator+(int a,int b)
{
   add a,b
   mov s_ret,a
}
```

编译器生成的汇编代码是这样：

```
push ebp
mov ebp , esp
add [ ebp + 8 ] , [ ebp + 12 ]
mov [ ebp + 16 ] , [ ebp + 8 ]
pop ebp
reti 8
```

其中有四条汇编语句是编译器生成的，另外两条是程序员写的。

由于 RPP 所有类和函数都是 public，故友元的含义已经变了，friend 和 static 是同义语，它们都表示该函数不会自动生成 this 引用，也就是该函数无法访问到本类的数据成员。

另外，operator 不是 RPP 的关键字，也就是说：

```
friend int operator+(int a,int b)
```

等价于

```
friend int +(int a,int b)
```

再看看如何调用 int.+(int,int)这个函数，表达式

```
1+2
```

会首先翻译为

```
int . + ( 1 , 2 )
```

再翻译成汇编代码

```
sub esp , 4
push 2
push 1
call [ & int +(int,int) ]
mov esi , esp
mov ebx , [ esi ]
add esp , 4
```

结合上面的栈空间安排表可以很清楚的看到函数的调用过程。

所有的寄存器也可以当做一个 int 类型的变量使用，比如想查看一下 esp 的值可以直接这样：

```
puts esp
```

想计算 ecx 与 edx 的商可以这样：

```
puts ecx/edx
```

判断寄存器是否为 0：

```
if(eax)
{
   …
}
```

但是不可以对寄存器赋值，下面这样是错误的：

```
eax=1+2
```

可以改为

```
1+2
mov eax,ebx
```

因为凡是返回 int 或者 bool 或者指针的表达式，编译器均会把这个返回值保存到 ebx 中。

bool 和 int 是一样的都是占用 4 个字节（C++的 bool 只占用一个字节）。

RPP 的汇编指令与 x86 大部分是相同的，只是另外增加了一些虚拟指令（也可以理解为伪指令），但是这些虚拟指令很容易转换成标准的 x86 指令。完整的指令列表请参考..\..\rpp\tvm.h。

9. 伪代码

可以把 RPP 当做伪代码写，下面就是《算法导论》开篇的插入排序代码（由于 RPP 数组从下标 0 开始而《算法导论》从 1 开始，故稍微修改了下）：

```
define ← =

void insertion_sort(rstr& a)
{
   for j ← 1 to a.count-1
      key ← a[j]
      i ← j-1
      while i>=0 && a.get(i)>key
         a[i+1] ← a[i]
         i ← i-1
      a[i+1] ← key
}

void main()
{
   rstr a="cab132"
   putsl a
   insertion_sort a
   putsl a
}
```

其中 for…to…也是一种语法糖

```
for i=1 to 10
```

等价于

```
for i=1;i<=10;i++
```

对于方法（函数）参数为空的情况，可以省略后面的括号，即

```
a.count
```

等价于

```
a.count()
```

10. 模板函数

RPP 的函数模板可以替换任何单词，跟宏差不多，比如有一个递归求和的函数(参考第 5 节)和一个递归求阶乘的函数：

```
int sum(int a)
{
   if a<=1
      return 1
   return sum(a-1)+a
}

int pro(int a)
{
   if a<=1
      return 1
   return pro(a-1)*a
}
```

发现这两个函数极其相似，那么可以这样定义一个模板函数：

```
int func<T>(int a)
{
   if a<=1
      return 1
   return func<T>(a-1) T a
}
```

像下面这样调用就可以了：

```
void main()
{
   puts func<+>(10)
   puts func<*>(10)

   puts func<+> 10
   puts func<*> 10
}
```

RPP 的模板函数十分强大，不仅支持模板函数作为类的成员，还支持模板动态生成：

```
void main()
{
   int* p=1p
   p.to<char*>.printl
   putsl typeof(p.to<char*>)

   A a
   puts a.func<2>
   puts a.func<5>
}

class A
{
   int m_a=2

   int func<T>
   {
      return m_a+T
   }
}
```

可以看到模板函数和普通函数使用方法完全相同，也不需要关键字 template，看起来更简洁一些。但是 RPP 的模板函数暂不支持类型推测，也就是说调用模板函数时后面的尖括号是必须的。

11. 宏

RPP 的 define 是在预处理阶段进行替换，功能比较弱，一般不需要使用。RPP 另有一个更好用的宏 mac，mac 宏属于类的成员，而不是作用于全局，这样可以更好地封装。比如：

```
mac fadd(a,b) a+b
```

等价于 C 语言的

```
#define fadd(a,b) ((a)+(b))
```

对于 mac 宏 RPP 会自动加上一些小括号避免优先级问题：

```
c=fadd(1,2)*fadd(1,2)
puts c
```

将输出正确的值 9。

另外 RPP 还支持另一种不自动加括号的宏，比如：

```
mac fadd2(a,b)
{
   a+b
}
```

这时用

```
c=fadd2(1,2)*fadd2(1,2)
puts c
```

得到的是 5，显然不是我们想要的结果。

但是这种宏另有不错的用法，请看下面的例子：

```
void main()
{
   pro=1
   for i=2;i<=10;i++
      pro+=i
   puts pro

   pro=1
   for i=2;i<=10;i++
      pro*=i
   puts pro
}
```

这是一个循环求和与循环求阶乘的程序，可以看到两段代码明显有相似之处，那么用 mac 定义一个宏：（宏中的花括号和分号不能省略）

```
mac fpro(T)
{
   pro=1;
   for(i=2;i<=10;i++)
   {
      pro T i;
   }
   puts(pro);
}
```

最后可以这样调用：

```
void main()
{
   fpro(+=)
   fpro(*=)
}
```

可以看到这与模板函数极其相似，与 C 语言的宏相比不需要写一堆的折行符号\，实际上 RPP 也没有折行符号。

另外，RPP 还支持一种拆分宏，比如：

```
#putsl(1,'abc',98)
```

宏展开后是这样：

```
putsl(1)
putsl('abc')
putsl(98)
```

那么连续插入 vector 或者 set 就很方便了：

```
vector<int> v
#v.push(1,99,2)
```

拆分宏的另一种用法请参考 example\11_3.h。

最新版 RPP 开始支持超级宏，变参宏可用超级宏实现，请参考 example\11_x.h。

12. 函数指针

RPP 使用一个比 C 语言更简单的函数指针语法，比如有一个这样的函数：

```
int fadd(int a,int b)
{
   return a+b;
}
```

C 语言通常是这样：

```
typedef int (*FADD)(int,int);
FADD p=fadd;
p(1,2);
```

或者

```
int (*p)(int,int)=fadd;
p(1,2);
```

可以看到 C 语言必须要指定函数指针的类型，但 RPP 可以直接这样调用：

```
int[&fadd,1,2]
```

或者

```
p=&fadd
int[p,1,2]
```

中括号左边表示返回值的类型（即使返回值为空也必须写 void），中括号里面的第一个参数是函数的地址，而后面的参数编译器会自动推断出类型。

&fadd 表示获取到函数的地址，这是一个静态地址，注意与标准 C 的区别，前面的取地址运算符是必须的。

由于 RPP 的函数指针只有一种类型，即

```
void*
```

那么

```
&fadd
```

得到一个类型为 void* 的指针常量。

如果有多个名字都是 fadd 的函数（重载），那就必须指定参数的类型：

```
&fadd(int,int)
```

如果类 A 需要获取到类 B 的某一函数的地址，那就必须指定类名：

```
&B.fadd(int,int)
```

或者

```
&B::fadd(int,int)
```

因为 RPP 中作用域运算符和成员运算符是等价的。

再看一个函数：

```
void func(int& a)
{
   ...
}
```

千万不要这样调用：

```
a=2
void[&func,a]
```

因为 func 的第一个参数 a 的类型是引用，正确的做法是：

```
a=2
void[&func,&a]
```

如果有另外一个函数：

```
void func2(int* a)
{
   ...
}
```

也可以这样调用：

```
a=2
void[&func2,&a]
```

13. 动态调用函数

在运行的过程中决定调用哪一个函数（运行时多态），这个特性在一些动态语言中十分常见。

还是拿上一节的函数举例：

```
int fadd(int a,int b)
{
   return a+b
}
```

那么可以根据字符串找到这个函数的地址：

```
p=findf("fadd")
```

然后调用

```
puts int[p,1,2]
```

或者合并上面两句：

```
puts int[findf("fadd"),1,2]
```

如果有函数重载的情况，就必须明确参数的类型：

```
p=findf("fadd(int,int)")
```

或者

```
p=findf("main.fadd(int,int)")
```

注意这种用法和上一节的函数指针有很大的区别，上一节的函数地址是在编译时确定的，而本节的函数地址是通过在运行时查找字符串获取到的。

14. 元函数

RPP 的编译器在运行期可以调用，因此可以动态生成代码。

例如：

```
void[meta('void lambda(){puts 123}')]
```

等价于

```
void[meta('(){puts 123}')]
```

可以看到元函数的定义方法和普通函数没有区别。

一般支持元编程的语言都可以用一行代码实现计算器，RPP 的版本是这样：

```
void[meta('(){putsl '+getsl+'}')]
```

或者可以更简单：

```
self('putsl '+getsl)
```

self 可以和自身进行交互，直接访问函数的局部变量或者参数：

```
void main()
{
   int a=0
   self('a=3')
   puts a
}
```

访问类成员变量也很容易：

```
class A
{
   int m_a=2

   func
   {
      self('puts this.m_a')
   }
}
```

另外，RPP 的元函数是线程安全的。

15. JS

RPP 1.87 暂时移除动态类型和 JS 支持。

16. 反射

RPP 拥有完整的反射和自省机制，解释器和 RPP 代码几乎可以融为一体：

```
import 'rpp.h'

main
{
    tasm* p=&main
    p->ptfi->name.printl
    p->ptfi->ptci->name.printl
}
```

上面的代码可以打印出自身的所属的函数和类，而打印自身的汇编代码以及表达式语句也很容易：

```
import 'rpp.h'

main
{
    r_print_asm('main')
    putsl
    r_print_sent('main')
}
```

遍历所有类请参考 example\16_3.h 以及 rsrc\rpp.h。

17. 闭包

RPP 支持静态闭包：

```
void main()
{
   int a=2
   void[lambda(){puts a}]
}
```

对于匿名函数访问外层变量，RPP 会调用赋值函数进行复制（与 C++11 的[=]类似）。

目前 RPP 的闭包有 3 个限制：
1. 不能在匿名函数中访问需要析构的外层变量。
2. 闭包需要访问的外层变量不能使用类型推断。
3. 闭包不是线程安全的。

RPP 的匿名函数写法和普通函数差不多：

```
lambda(int a,int b){}
lambda(int,int a,int b){}
```

上面第一个函数的返回值为空，带两个整型参数。第二个函数返回值为 int，带两个整型参数。如果第一个参数只有类型，而没有变量名，编译器理解为这是一个返回值类型。

18. 多重继承

RPP 支持多重继承，不过 RPP 的继承方式很特别，简单而有效：

```
class A
{
   int m_a

   fa
   {
      m_a=1
   }
}

class B:A
{
   fb
   {
      m_a=2
   }
}
```

然后这样使用：

```
B b
b.fa
puts b.m_a
b.fb
puts b.m_a
```

可以看到 B 类继承了 A 类的数据成员 m_a 和函数成员 fa。对于 B 类继承自 A 类，RPP 只是简单地拷贝代码，因此，RPP 的继承对程序员是透明的：

```
class B
{
   int m_a

   fa
   {
      m_a=1
   }

   fb
   {
      m_a=2
   }
}
```

就像上面这样，直接把 A 类的所有代码拷贝到 B 类的头部。这也是 RPP 的哲学：熟练运用“Ctrl+C”和“Ctrl+V”可以省下人生 80%的时间。

显然，RPP 的子类构造时不会自动调用父类的构造函数，且不允许父类和子类存在同名且参数类型和个数都相同的函数。

RPP 目前支持 3 种继承：
*模板继承模板 A<T>:B<T>
*模板继承非模板 A<T>:C,D
*非模板继承非模板 E:C

暂不支持继承模板实例 C:A<int>

多重继承的例子请参考：
example\18_2.h

19. 变参函数

RPP 使用中括号进行变参函数调用。

求两个数的和：

```
sum[1,2]
```

求三个数的和：

```
sum[1,2,3]
```

实际上 RPP 会自动把参数的个数作为参数传递过去，即

```
sum[1,2]
```

等价于

```
int[&sum,2,1,2]
```

目前 RPP 的可变参数用起来虽然简单，写起来却比较痛苦：

```
int sum(int count)
{
    int* p=&count+1
    int ret=0
    for i=0;i<count;i++
        ret+=*p
        p++
    *p=ret

    mov ecx,4
    imul ecx,count
    add ecx,4
    pop ebp
    add esp,sizeof(s_local)
    mov eax,[esp]
    _reti(eax,ecx)
}
```

类成员函数使用可变参数的例子是：

```
example\19_2.h
```

20. 默认参数

举例说明：

```
int func(int a,int b=a)
{
   return a+b;
}

void main()
{
   puts func(2)
   puts func(1,2)
}
```

有必要解释一下它的工作原理，对于上面的 func 函数，编译器将自动生成两个函数：

```
int func(int a)
{
   int b=a;
   return a+b;
}

int func(int a,int b)
{
   return a+b;
}
```

显然，RPP 的默认参数比 C++更灵活，因为后面的参数不仅可以访问到前面的参数，还可以访问到类的成员变量：

```
class A
{
   int m_a

   func(int a,int b=m_a)
   {
      ...
   }
}
```

21. 多线程

C++通常是这样封装线程的：

```
class A
{
   int m_a;

   static void* thread(void* param)
   {
      A* pthis=(A*)param;
      pthis->m_a=2;
      ...
   }
}
```

RPP 则是这样：

```
class A
{
   int m_a

   static void thread(A& this)
   {
      this.m_a=2
      ...
   }
}
```

或者可以更简单地：

```
class A
{
   int m_a

   thread
   {
      m_a=2
      ...
   }
}
```

因为不加 static 编译器会自动生成一个 this 引用。

创建线程直接调用函数即可：

```
A a
p=rf.create_thr(&A.thread,&a)
```

如果需要等待线程退出则：

```
rf.wait_thr(p)
```

如果不想把线程封装进类可以直接这样：

```
rf.create_thr(&thread)
```

22. 本地调用

由于 RPP 的所有数据类型和 C/C++二进制兼容,因此 RPP 调用外部函数十分方便。

下面是解释模式的调用方法:

```
stdcall["MessageBoxA",0,"abc","123",0]
```

注意 RPP 的字符串常量统一使用 UTF8, 调用 UTF16 版本的 API 需要转换编码:

```
stdcall["MessageBoxW",0,utf16c("abc"),utf16c("123"),0]
```

调用外部的 cdecl 函数:

```
cdecl["strlen","abc"]
```

解释模式调用 strlen 也可以这样:

```
int strlen(char* p)
{
    sub esp,4
    push p
    calle c_strlen
    mov s_ret,[esp]
    add esp,4
}
```

编译模式的调用方法:

```
int strlen(char* p)
{
    push p
    rn invoke strlen
    add esp,4
    mov s_ret,eax
}
```

JIT 模式的调用方法:

```
int strlen(char* p)
{
    push p
    calle "strlen"
    add esp,4
    mov s_ret,eax
}
```

对比以上三种模式可以很清楚地看到 RPP 和 C++传递返回值的方式有区别，C++通过 eax 返回，RPP 通过栈返回，因此二者相互调用时要注意转换。

对于 C++调用 RPP，需要增加一个胶水函数保护 ebx、esi、edi（标志寄存器和 xmm 寄存器无需保护），比如：

```
void cpp_func()
{
    push ebx
    push esi
    push edi
    call rpp_func
    pop edi
    pop esi
    pop ebx
}
```

对于 RPP 调用 C++，则直接调用即可。

JIT 模式时 RPP 和 C++共享整个进程空间，二者相互调用十分简单，因此推荐使用 JIT 模式。

另外，如果 DLL 没有加载到当前的进程空间，请先调用 LoadLibraryA(W)。

23. 类型转换

RPP 没有强制转换，可以通过函数转换类型：

```
2.touint
```

或者

```
2.to<uint>
```

或者

```
uint(2)
```

把有符号整数 2 转换成无符号整数。

整数和字符串之间的相互转换是很方便的：

```
2.torstr
"123".toint
```

RPP 系统库只定义了 char 和 int 的相互转换：

```
char ch=`a
putsl ch.toint
```

将输出 97。

```
char ch=-1
putsl ch.toint
```

将输出 255。

注意和标准 C 的区别，RPP 将 char 转换为 int 不会进行符号扩展。如果需要 char 和 uint 的相互转换请修改 rsrc\char.h 这个文件。

RPP 支持自动类型转换：

```
void func(uint a)
{
    ...
}
```

那么

```
func(2)
```

会自动替换为

```
func(uint(2))
```

所以，请小心定义拷贝构造函数。

另外，和 Java 一样，RPP 推荐尽量少使用无符号数。

24. 动态数组

RPP 里类名和对象名不可以同名，下面这样是错误的：

```
myclass myclass;
```

汇编的关键字不可以用来命名变量，但可以命名函数，比如 vector 类中有一个 push 函数，和汇编关键字 push 同名，下面这样没有问题：

```
vector<int> a
a.push(2)
```

RPP 用 rbuf 代替 STL 的 vector，用 rstr 代替 string，用法大致相同。push 就是 vector 的 push_back，而 cstr 就是 string 的 c_str。

打开 rsrc\basic.h 可以找到：

```
#define vector rbuf
#define string rstr
```

要注意的是 rbuf 的 size 方法和 count 方法是不同的：

```
vector<int> a
a.push(2)
putsl a.size
putsl a.count
```

输出是：

```
4
1
```

即 size 返回所有元素占用的字节数，而 count 返回元素个数，这和 STL 是有区别的。

25. 多参数组

RPP 没有静态数组，新版本也不再提供数组定义的语法糖，仅使用多参数组（数组运算符带多个参数）来实现多维数组：

```
import "rbufm.h"

void main()
{
   rbufm<int> arr(5,5)

   for i=0;i<5;i++
      for j=0;j<5;j++
         arr[i,j]=i
         arr[i,j].print
}
```

下面是三维数组的例子：

```
import 'rbufm.h'

void main()
{
   rbufm<int> arr(5,3,4)

   for i=0 to 4
      for j=0 to 2
         for k=0 to 3
            arr[i,j,k]=j
            arr[i,j,k].print
}
```

需要注意的是，如果将数组作为函数参数传递，会传递数组的拷贝，也就是说，形参数组和实参数组互不影响。如果形参数组和实参数组需要共享内存，那么可以传递引用：

```
void func(rbufm<int>& arr)
{
   ...
}
```

当然也可以使用嵌套的模板来实现多维数组：

```
void main()
{
   rbuf<rbuf<rbuf<int>>> arr

   arr.alloc(5)
   for i in arr
      arr[i].alloc(3)
      for j in arr[i]
         arr[i][j].alloc(4)

   for i in arr
      for j in arr[i]
         for k in arr[i][j]
            arr[i][j][k]=j
            arr[i][j][k].print
}
```

以上代码定义了一个 5 层 3 行 4 列的三维数组，其中的 for...in... 也是一种语法糖：

```
for i in arr
```

等价于

```
for(i=0;i<arr.count;i++)
```

26. 重载

不同类型指针作为函数参数传递时不需要强制转换，因此仅有指针类型不同的函数不能重载，下面的写法是有歧义的：

```
void func(int* a)
{
   ...
}

void func(char* a)
{
   ...
}
```

不同类型的引用可以重载：

```
void func(int& a)
{
   ...
}

void func(char& a)
{
   ...
}
```

相同类型的引用和非引用可以重载：

```
void func(int& a)
{
   ...
}

void func(int a)
{
   ...
}
```

下面是自定义结构体的例子：

```
class A
{
   int* m_p

   A()
   {
      m_p=r_new<int>5
   }

   A(A& a)
   {
      m_p=r_new<int>5
      for i=0 to 4
         m_p[i]=a.m_p[i]
   }

   ~A()
   {
      if m_p!=null
         r_delete<int>m_p
         m_p=null
   }

   operator=(A& a)
   {
      this.~A
      this.A(a)
   }
}
```

自定义类型必须提供两个拷贝构造函数（一个引用一个非引用），如果没有非引用版本编译器会自动生成一个和引用版本函数体完全相同的函数，赋值函数也是一样。（多个参数的构造函数不会自动增加非引用版本）

另外，默认参数的重载规则和 C++一样，不再赘述。

27. 指针和引用

RPP 里引用不是“别名”，而是地址，因此千万不要这样写：

```
a=2
int& b=a
```

正确的做法是：

```
a=2
int& b
lea b,[ebp+s_off a]
```

其中 s_off 表示取得局部变量的偏移，这样的话 a 与 b 共享同一段内存。

```
a.print
b.print
```

将输出两个 2。

一般不会使用引用作为局部变量，而是使用引用作为函数参数：

```
func(a)
a.print
```

函数 func 的定义是：

```
void func(int& n)
{
   n=5
}
```

指针的用法和标准 C 差不多：

```
a=3
p=&a
a.print
p->print
```

对于二重指针（或以上）则不可以使用类型推断：

```
a=4
p=&a
int** pp=&p
```

对于三重指针（或以上）RPP 也没有定义语法糖，因此需要使用嵌套的模板来定义：

```
rp<rp<rp<int>>> ppp=&pp
puts *(*(*ppp))
```

由此可以看到 RPP 的指针实际上是模板类的一个实例，具体情况请参考：

```
rsrc\rp.h
```

28. 全局变量

全局变量定义的时候不能使用类型推断，而且必须以 g_开头：

```
int g_a=2
rbuf<char> g_b(3)

void main()
{
   putsl g_a
   b=g_b
   b.count.printl
}
```

29. 成员变量

举例说明：

```
class A
{
   int m_a=2
   int m_b(3)

   A()
   {
   }
}
```

即对于类的成员变量（数据成员），可以直接在定义处初始化，上面的例子等价于：

```
class A
{
   int m_a
   int m_b

   A()
   {
      m_a=2
      m_b=3
   }
}
```

即编译器会自动在该类的每一个构造函数的头部加上这么一个初始化语句。

另外，RPP 将类的成员变量统一按 1 字节对齐：

```
class A
{
   char m_a
   int m_b
}
```

这样的话

```
   sizeof A
```

将返回 5。注意这样写的话在 x86 上不会有问题，但移植到 arm 上就有问题了，故不推荐这种写法。

30. 局部变量

和 javascript 一样，RPP 的局部变量在整个函数都是可见的，比如：

```
for(int i=0;i<10;i++)
{
   ...
}
if(i>=10)
   ...
```

重复定义两个名字和类型都相同的变量也是允许的：

```
vector<int> v
v.push(2)
v.count.print

vector<int> v
v.count.print
```

输出是

```
1
0
```

因为 RPP 中定义变量会先析构后构造，这表示将一个变量重新初始化。故 RPP 的类必须支持空构造函数（即使没有编译器也会自动生成一个），并且需要在析构的时候判断是否已经析构：

```
class A
{
   int* m_p

   A()
   {
      m_p=r_new<int>5
   }

   ~A()
   {
      if(m_p!=null)
         r_delete<int>m_p
         m_p=null
   }
}
```

另外，可以使用赋值运算符进行类型推断：

```
a=1+2
```

等价于

```
auto a=1+2
```

等价于

```
int a=1+2
```

31. 常量

RPP 目前还不具备 perl 那种强大字符串处理能力，不过也向它学习了一些经验。对于双引号扩起来的字符串，编译器并不会总是提供语法糖（请参考 No.6），实际上它还是标准 C 的以 0 结尾的 ASCII 串。所以，为了方便处理，RPP 还提供了一个用单引号字符串：

```
'123'
```

等价于

```
rstr("123")
```

因此，单引号字符串完全可以当做内置字符串使用，转换成 C 字符串可以这样：

```
'123'.cstr
```

连续两个反斜杠表示从当前位置到本行末均为单引号字符串，并且不进行转义：

```
s=\\abc'123
```

等价于

```
s='abc\'123'
```

主要用于字符串内部转义字符和引号很多的情况。

RPP 没有字符常量，但是提供一个反单引号：

```
`a
```

等价于整数

```
97
```

表示单个字符的方法是：

```
r_char("a")
```

或者

```
r_char('a')
```

如果整数比较长可以用下划线分隔：

```
1_0000_0000
```

等价于

```
100000000
```

以 0x 开头的常量表示 16 进制整型常量，以 0b 开头的常量表示二进制整型常量：

```
0xa
```

等价于

```
0b1010
```

等价于

```
10
```

双精度浮点(double)常量目前只有小数写法，比如：

```
0.5
```

32. 控制结构

选择结构和大多数语言一样：

```
if ...
   dosomething
elif ...
   dosomething
else
   dosomething
```

对于省略花括号的情况，编译器会自动加上花括号，对于花括号没有单独占一行的情况，编译器会自动换行，要注意编译器提示的行号是自动处理之后的行号。

打开 rsrc\basic.h 可以找到：

```
#define elif else if
```

即 elif 是一个宏。

条件结构是这样：

```
switch ...
{
   case 1:
   {
      ...
   }
   case 2:
   {
      ...
   }
   default:
   {
      ...
   }
}
```

可以省略花括号和冒号：

```
switch ...
case 1
   ...
case 2
   ...
default
   ...
```

RPP 的条件结构不需要 break，default 是可选的，但如果有必须排在最后。（实际上目前 RPP 还没有进行 switch 优化）

带增量的循环结构：

```
for i=1;i<10;i++
    ...
```

不带增量的循环：

```
for i<10
    ...
```

死循环有几种写法：

```
for true
    ...
```

或者

```
for 1
    ...
```

或者

```
for
    ...
```

打开 rsrc\basic.h 可以找到：

```
#define while for
```

即 while 和 for 是同义语，是否赋初值和执行增量仅取决于条件表达式里面有没有分号。

另外，RPP 有两种 continue，一种是不执行增量直接进行条件判断：

```
continued
```

另一种是执行增量后再进行条件判断：

```
continue
```

至于 break 和大多数语言一样，不再赘述。（是否要增加关键字用于跳出多重循环是一个值得讨论的问题）

和 Go 语言一样，RPP 也抛弃了 do...while，下面是 do...while 的等价写法：

```
for
{
    ...
    ifn(i<10)
        break
}
```

或者

```
for
{
    ...
    until(i>=10)
}
```

其中 ifn 是语法糖：

```
ifn(i<10)
```

等价于

```
if(!(i<10))
```

until 是宏：

```
until(i>=10)
```

等价于

```
if(i>=10)
{
    break;
}
```

RPP 的 goto 和 jmp 是同义语，下面是一个死循环：

```
F:
    ...
    jmp F
```

如果你确定某个函数编译器没有自动换行，可以直接在 jmp 后面加一个整数表示直接跳转到该行，下面是直接跳转到第 7 行：

```
...
jmp 7
```

进行条件跳转也是很容易的：

```
a<b
jebxz G
    putsl "a less than b"
G:
```

等价于

```
if(a<b)
    putsl "a less than b"
```

注意只有下面几种类型的表达式可以作为条件表达式：
（如果使用其它类型作为条件表达式会在运行时出错）

1. int
2. bool
3. 指针

33. 模板类

模板类与 C++的区别是构造函数和析构函数后面必须有尖括号和模板参数：

```
class A<T>
{
   T m_a

   A<T>
   {
   }
}

void main()
{
   A<int> a
   A<A<int>> b

   a.m_a=2
   a.m_a.print
   b.m_a.m_a=3
   b.m_a.m_a.print
}
```

目前 RPP 的模板有 3 个限制：
1. 不支持模板元编程。
2. 不支持模板默认参数。
3. 不支持变参模板。

关于模板类嵌套请参考：

```
example\33_2.h
```

34. 包

RPP 中：

```
import
```

等价于

```
#include
```

使用方法：

```
import "A.h"
```

或者

```
import 'A.h'
```

或者

```
import A.h
```

注意不要这样写：

```
import <A.h>
```

因为 RPP 不支持尖括号文件名，用双引号括起来的文件名 RPP 会在当前目录和编译器目录搜索。另外，和 D 语言、objective-c 一样，RPP 会自动处理重复包含的问题。

RPP 支持 mixin 和递归引用，类和函数均不需要声明即可使用，也就是说 RPP 没有.cpp 文件，只有.h 文件（这样就不需要向前声明）。

（v1.87 的“包”仍在开发中，目前只有 v1.1 支持“包”）

打开 rsrc\basic.h 可以找到：

```
#define namespace friend class
```

清楚地看到 RPP 的命名空间不过是一个友元类，或者可以称为静态类，即所有函数都没有 this，下面举例说明：

```
namespace MY
{
   //RPP 没有枚举变量，只有枚举常量
   enum
   {
      c_a=2
      c_b=4
   }

   void func()
   {
      puts c_a
   }
}

void main()
{
   MY.func
   MY::func
   puts MY.c_a
   puts MY::c_b
}
```

命名空间嵌套和类嵌套请参考 example\34_2.h。

35. 堆

RPP 的 new 和 delete 既不是运算符，也不是关键字，而是模板函数：

```
p=r_new<int>(5)
```

表示从堆中分配 5 个 int。

释放刚才分配的内存：（不需要 delete [] p）

```
r_delete<int>(p)
```

一般来说，不需要使用 new 和 delete，（一个很简单的原因，不用 new 就几乎不可能出现内存泄露），推荐使用 rbuf 来进行内存分配：

```
rbuf<int> a(5)
```

因为 rbuf 会在析构时自动释放内存，当然也可以手动释放：

```
a.free
```

可以使用数组运算符或者

```
a.begin
```

或者

```
a.m_p
```

访问刚才分配的内存。

如果只需要一个对象，Java 通常是这样：

```
B b=new B();
```

而 RPP 则这样：

```
B b
```

一个对象通常在栈中分配就行了。

36. 库

RPP 内核非常小，没有内置数据结构，甚至可以认为 RPP 没有内置数据类型，所有类型均在外部定义，这样做的好处是可移植性强（移植到 64 位改动很小）。下面是以模板类的形式提供的几种数据结构：

名称	用途
rbuf	动态数组，栈
rstr	字符串
rstrw	utf16 字符串
rlist	双向链表，队列，栈
rset	红黑树
rdic	字典
rhash	哈希表

另外，rsrc\ralgo.h 提供了快速排序、二分查找、split。下面还有一些封装好的类（示例代码请参考 example\36_x.h）：

名称	用途
rflie	文件操作
rdir	遍历目录
rsock	TCP 套接字
rmutex	互斥体
rcode	编码转换

37. 编译和链接

RPP 支持多种运行模式，默认是解释运行（不产生多余文件），直接在命令行敲入（如果不想每次敲命令可以把 rpp.exe 和.h 后缀文件类型关联起来）：

```
rpp example\1.h
```

JIT 运行：

```
rjit example\1.h
```

编译生成 exe 也很简单，先 cd 到 RPP 主目录，然后敲入命令：

```
build_run example\1.h
```

以上命令会生成 example\1.exe 并运行，默认是动态链接到 C 运行库 msvcr100.dll，因此生成的 exe 会依赖于 VS2010 运行库。您也可以修改 example\run.h 让 GoLink.exe 链接 msvcrt.dll，这样生成的 exe 可移植性更强，但无法调用_fseeki64 等 64 位文件操作函数。

只运行不产生多余文件：

```
run example\1.h
```

只生成 exe 不运行：

```
build example\1.h
```

生成 asm 源文件的方法是先 cd 到 RPP 主目录，然后敲入：

```
rpp -win example\1.h
```

以上命令会生成 example\1.asm ，这样就可以拿着这个源文件用 nasm 去汇编和链接了（生成静态链接库和动态链接库的必须步骤）。

RPP 翻译成 nasm 符号的规则是将所有非数字和非字母转换为 16 进制表示：

```
int.+(int,int)
```

会翻译为

```
int2E2B28int2Cint29
```

编译生成 DLL 需要先修改 asm 源文件：

```
   entry DllEntry

   proto cdecl,test

   [section .text]

   proc DllEntry,ptrdiff_t hin,size_t rea,size_t res
   locals none
      mov eax,TRUE
   endproc

   proc test
   locals none
      invoke MessageBox,NULL,cont,cont,MB_OK
      xor eax,eax
   endproc

   [section .data]
      cont:declare(NASMX_TCHAR) NASMX_TEXT('hello'),0x0
```

以上代码导出了符号 test，然后敲入命令：

```
   nasm -f win32 1.asm -o 1.obj
   GoLink.exe  /dll  /export  _test  /entry  _main  1.obj
kernel32.dll user32.dll
```

成功后会生成 1.dll，这时即可在其他语言中调用 test 函数，下面以 nasm 为例：

```
   entry demo

   import cdecl,test

   [section .text]

   proc demo,ptrdiff_t argcount,ptrdiff_t cmdline
   locals none
      invoke test
      invoke ExitProcess,NULL
   endproc
```

38. 裸奔

RPP 的跨平台能力十分强悍，支持 grub 从 U 盘或者硬盘引导，不需要操作系统也可以运行，步骤如下：

1. cd 到 RPP 主目录下的 nasm 目录
2. 敲入 bin cell.h （成功后会生成 cell.bin）
3. 修改 menu.lst 将引导文件改为 cell.bin
4. 使用 bootice（或其它工具）制作 grub4dos 引导扇区
5. 将 cell.bin、menu.lst、grldr 三个文件拷贝至 U 盘或硬盘根目录
6. 推荐先使用虚拟机进行测试

引导之后已经进入了保护模式，堆默认分配 64M，栈 1M，请查看 rsrc 下的源码了解具体情况。

RPP 并没有定义所有的 nasm 指令，但可以使用 rn 伪指令将文本直接输出到 nasm，比如：

```
rn mov dx,0x1f7
rn mov al,0x20
rn out dx,al
```

这样配合《自己动手写操作系统》和《30 天自制操作系统》就可以 DIY 自己的操作系统了。

39. 图形界面

RPP 支持 cocos2dx 绑定，开发跨平台小游戏十分简单，windows 下输入命令：（仅 v1.83 及以下版本支持，新版本正在重构 cocos2dx 绑定）

```
rcc cocos\flappy.h
```

这是一个 flappy bird 山寨版的例子，注意 cocos2dx 需要 VS2012 运行库和最新的 opengl 驱动。

如果需要指定 windows 窗口大小请输入：

```
rcc cocos\flappy.h 800 600
```

不显示控制台：

```
rcc cocos\flappy.h 800 600 -noconsole
```

android 下安装 cocos\rcc.apk 即可，如果需要打包自己的 apk 需要先把 libcocos2dcpp.so 解压出来，然后下载 cocos2dx 3.0，再使用 eclipse 或者 ant 打包。

40. 文言文

RPP 的标识符支持中文，源文件支持 GBK、Unicode(UTF16 little endian)以及 UTF8 三种编码。下面是中文编程的第一个例子，打印“你好”：

```
主即曰'你好'也
```

运行这个例子之前请先设置控制台（因为 RPP 的字符串常量使用 UTF8）：

```
chcp 65001
```

然后在命令行标题栏上点击右键，选择"属性"->"字体"，将字体修改为 True Type 字体"Lucida Console"，再点击确定将属性应用到当前窗口。

然后删除 rinf\optr.txt，再将 rinf\optr2.txt 重命名为 rinf\optr.txt，最后反注释 rsrc\basic.h 中的第一行：

```
import "chs.h"
```

下面再看一个递归求和的例子：

```
主即曰和 100 也

数和 数甲 即
若甲 小于 2
    归 1
归甲 加和 甲 减 1
也
```

注意有些地方要用空格分隔，更多的例子请参考：

```
example\40_x.h
```

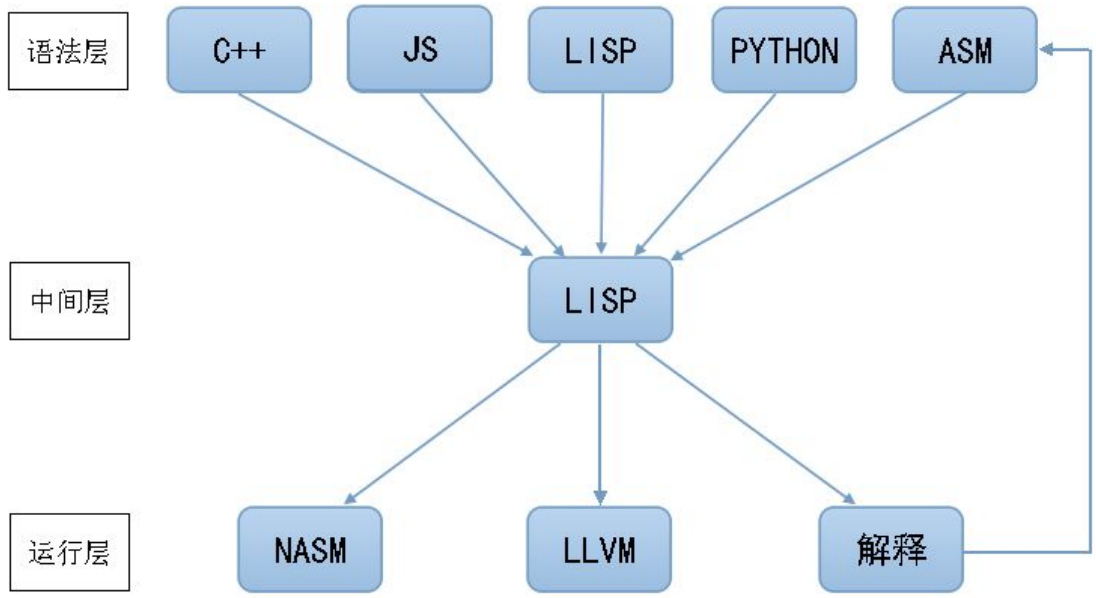
41. 语言特性

RPP 的 5 种运行模式支持的特性有一些区别：

	JIT	解释	编译	裸奔	cocos2dx 绑定
动态调用函数	√	√	×	×	√
元函数	×	√	×	×	√
反射和自省	×	√	×	×	√
惰性求值	×	√	×	×	√
JS	×	√	×	×	√
闭包	√	√	√	√	√
可变参数	√	√	√	√	√
多线程	√	√	√	×	√
本地调用	√	√	√	×	×
静态和动态链接	×	×	√	×	×

表中打叉的选项也并非绝对，有些特性仍然可以由用户手动实现。另外，没有列入表中的特性表示 5 种运行模式都支持。

RPP 的整体设计图：



42. 函数对象

RPP 不支持重载小括号，但是可以重载中括号实现 C++的函数对象（仿函数）：

```
void main()
{
   A<int,int> a=lambda(int,int n){return n*n}
   puts a[3]
}

class A<T1,T2>
{
   void* m_func

   =(void* p)
   {
      m_func=p
   }

   T1 [](T2 n)
   {
      return T1[m_func,n]
   }
}
```

43. 效率

运行效率：
RPP 的内部结构和 C++大致相同，所以理论上 RPP 可以达到和 C++一样的运行速度，目前 RPP 已开启汇编级内联展开和模式匹配优化，在编译模式或 JIT 模式下的运行速度是 C++的 1/1.8（真实速度应该是 1/3），解释模式是 C++的 1/10.7。目前的性能与 luaJIT 相当接近，比 TCC 略快。

编译效率：
RPP 对 C++的一些复杂语法进行了简化，单纯的编译速度比 C++略快。不过 RPP 在解释运行的时候并不会一次编译整个程序，而是在函数需要运行的时候才进行即时编译，所以综合编译速度比 C++快一个数量级（仅与 VC++和 G++对比）。目前 RPP 的编译模式使用 NASM 作为后端，所以编译生成 EXE 的速度较慢，但解释模式和 JIT 模式的编译速度比较快。

开发效率：
RPP 不需要建立工程或者 makefile，直接新建一个文本文件（任意后缀）即可开始编码。它也可以很好地与神器 Visual Assist 或者 Source Insight 进行配合，使用 Visual Studio 实现代码智能提示和自动补全。（请参考视频演示）

调试方法：
RPP 目前没有配套的调试环境。如果语法通过而逻辑出错，建议先使用注释和输出语句来确定出错的位置，然后使用第 16 节的反射方法打印出相关函数的表达式语句和汇编代码。当然也可以使用 OllyDbg 进行汇编级调试。

下面是 RPP 的性能测试数据：（奔腾 1.86GHZ，测试 3 次取平均值）

	运行耗时	对比 C++
RPP	2277 ms	10.7/1
RPP JIT	374 ms	1.8/1
RPP NASM	374 ms	1.8/1
lua 5.1	24710 ms	111/1
luaJIT 2.0	266 ms	1.2/1

RPP 测试代码：

```
sum=0
for i=0;i<=10000;i++
   for j=0;j<=10000;j++
      sum+=i
putsl(sum)
```

lua 测试代码：

```
i=0
while i<=10000 do
   j=0
   while j<=10000 do
      sum=sum+i
      j=j+1
   end
   i=i+1
end
print(sum)
```

lua 使用 for 循环测试：

```
sum=0
for i=0,10000 do
   for j=0,10000 do
      sum=sum+i
   end
end
print(sum)
```

使用 for 循环时 luaJIT 2.0 耗时几乎一样，但 lua 5.1 性能提升了 2 倍。不过 lua 的 for 循环无法在循环过程中改变循环变量的值，并不具有通用性。

（目前官方只提供了整数的优化，如需浮点优化请修改 rinf\match.txt）

44. 惰性求值

RPP 的所有运算符都是函数，而且 RPP 会在非引用返回值传递引用的情况下增加临时变量，所以不要依赖函数的传参顺序。

如果 p 是一个整型指针，那么下面的写法在 RPP 中是错误的：

```
if p!=null && *p==2
   ...
```

可用等价的写法代替：

```
if p!=null
   if *p==2
      ...
```

或者

```
if p==null
   return false
if *p==2
   ...
```

假如 v 是一个整型数组，下面的写法同样是错误的：

```
if v.empty || v[0]!=5
```

可用等价的写法代替：

```
if v.empty || v.get(0)!=5
```

RPP 1.87 暂时移除惰性求值。