目录

二、Using Object

2.1 string

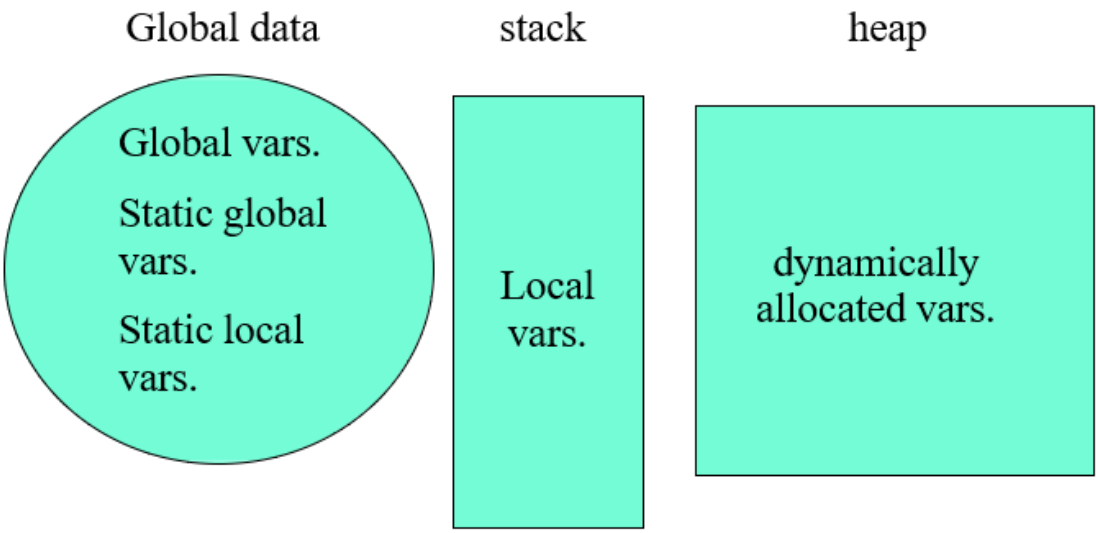
```
#include <string>
//定义
string str1 = "Hello", str2, str3;
//输入、输出
cin >> str1;
cout << str1;
//运算
str2 = str1;
str3 = str1 + str2;
str1 += str2;
str1 += "lalala";
//构造函数
string s1("Hello"), s2("World");
string(const char *cp, int len);
string(const string& s2, int pos);
string(const string& s2, int pos, int len);
//子串
substr(int pos, int len);
//修改串
assign();
insert(const string &s, int len);
insert(int pos, const string &s);
erase();
append();
replace();
//寻找串
find();
```

2.2 File I/O

```cpp
#include <fstream>
ofstream File1("C:\\test.txt");
File1 << "Hello World" << endl;

ifstream File2("C:\\test.txt");
string str;
File2 >> str;
```

2.3 内存结构



全局变量**Global Vars**：

(1)在函数外定义

(2)可以通过**extern**，分享给不同的***.cpp**文件

extern：

(1)声明，在整个工程中会有这样一个变量

(2)声明了变量的类型、名称

static：

(1)禁止被其他***.cpp**文件访问

(2)包括**static**变量、**static**函数

static local var：

(1)保存了在不同次访问函数时，需要被保存的变量

(2)在第一次访问函数时，变量被初始化

2.4 Pointers to Objects

```cpp
string s = "hello";
string *ps = &s;
```

&：取地址

```cpp
ps = &s;
```

*：得到对象

```
(*ps).length();
```

->：调用函数

```
ps->length();
```

注：

```
string s; //会将对象s进行初始化
string *ps; //ps指向的对象不一定存在
```

2.5 动态分配内存

动态分配内存：保存在堆空间

```
//分配内存：new--自动调用结构体的初始化操作
//分配内存时,系统会默认在内存的前面加上一些描述信息,以便于系统后期回收
new int;
new Stash;
new int[10];
//删除内存：delete--调用每个对象的析构函数
delete p; //删除一个对象
delete[] p; //删除数组
```

动态数组：

```
int *psome = new int[10];
delete[] psome;
```

注意：

```
int *pArr = new int[10];
pArr++;
delete[] pArr; //这句话是不对的,pArr指向的不仅仅是自身,还有堆分配的内存的相关信息
```

要求：

(1)不要**delete**不是被**new**分配的内存

(2)不要**delete**同一个内存两次

(3)如果**new []**，要用**delete []**

(4)可以**delete**一个空指针

2.6 引用 Reference

用法：对一个已经存在的对象，生成一个新的名字

```
//定义,定义后,访问r <==> 访问c
//如: r='t' <==> c='t'
char& r = c;
//应用
void func(int &x); //在func函数中,对x的修改会返回
```

例:

```
string s1("Hello"), s2("World");
string& rs1 = s1;
rs1 = s2;//这句话表示将rs1赋值为"World",而非将rs1重新绑定到s2上
```

Pointers vs References:

(1)**Pointers**:

(a)可以是**null**

(b)不依赖于存在的对象

(c)可以修改指向不同地址

(2)**References**:

(a)不能是**null**

(b)依赖于已经存在的对象

(c)定义后，不能指向其他对象

Referencce的限制:

(1)不能引用一个引用对象

(2)不能用指针指向引用对象：**int&* p**

(3)可以引用指针：**int*& p**

(4)不能定义一个数组的引用

2.7 常量 const

```
//定义,定义后x不能修改
const int x = 123;
```

使用**const**代替宏定义常量，优点:

(1)有作用域

(2)宏是强替换，而**const**是一个变量

编译器对**const**的处理:

(1)编译器会尽量避免为**const**变量分配内存，而是放入符号表中

(2)**extern**的**const**变量，会被分配内存

```
const int bufsize=1024;
extern const int bufsize;
```

2.7.1 常量数组

```
const int i[] = { 1, 2, 3, 4 };
float f[i[3]]; // Illegal

struct S {
    int i, j;
};
const S s[] = { { 1, 2 }, { 3, 4 } };
double d[s[1].j]; // Illegal
```

2.7.2 Pointers and const

```
//指针本身不被更改:q is const
char * const q = "abc";
*q = 'c'; //OK
q++; //Error

//指针指向的内容不被更改:(*p) is const
const char *p = "ABCD";
*p = 'b'; //Error;

//例
string p1("Fred");
const string* p = &p1; //(*p)不能更改
string const* p = &p1; //(*p)不能更改
string *const p = &p1; //p不能更改
```

Pointers and constants:

	int i	**const int ci = 3;**
int * ip;	ip = &i;	ip = &ci; //不合法
const int * cip;	cip = &i;	cip = &ci;

```
*ip = 54; //永远是合法的,因为ip指向的是int
*cip = 54; //不可能是合法的,因为ip指向的是const int
```

2.7.3 String Literals

```
char *s = "Hello World";
//这里s指向了一个字符串常量
//但是编译器允许不写const char* s
//*s不能修改
char s[] = "Hello World";
//如果需要修改字符串,需要保存到数组之中
```

2.7.4 常量与非常量的转化

非常量 ==> 常量：可以直接转化

```
void f(const int *x);
int a = 15;
f(&a);
```

常量 ==> 非常量：使用**const_cast**关键字

2.7.5 传递并返回参数的地址

(1)传递一整个对象，代价会非常大

(2)我们应该传递对象的地址，而非对象

(3)但是此时，会导致函数可以修改原始对象

(4)事实上，如果需要给函数传递一个地址，我们需要将其设置为**const**

```
void f(const int *x);
```