目录

[前言 2](#_Toc8195)

[工程项目示例 2](#_Toc20479)

[一. STL(Standard Template Libray) 3](#_Toc15373)

[二. 函数 3](#_Toc31118)

[2.1 函数模板 3](#_Toc6907)

[2.1.1 函数模板特化 4](#_Toc19479)

[2.2 重载函数 4](#_Toc20234)

[2.3 内联函数 4](#_Toc16395)

[2.3.1 普通函数 4](#_Toc2410)

[2.3.2 内联函数 5](#_Toc31317)

[三. 面向对象 6](#_Toc6638)

[步骤 6](#_Toc6086)

[示例 6](#_Toc11585)

[一. 数据类型 8](#_Toc4345)

[1.1 进程内存分配 8](#_Toc17864)

[1.2 变量的存储类型 8](#_Toc25125)

[1.3 基本数据类型 9](#_Toc3228)

[1.3.1 整型 9](#_Toc16060)

[1.3.2 字符型 10](#_Toc17213)

[1.3.3 浮点型(实型) 11](#_Toc5833)

[1.4 构造数据类型 11](#_Toc9195)

[1.4.1 数组类型 11](#_Toc9317)

[1.4.2 结构体类型 12](#_Toc11725)

[1.4.3 枚举类型 12](#_Toc10689)

[1.4.4 共用体(联合类型) 13](#_Toc847)

[1.5 指针类型 13](#_Toc3223)

[1.5.1 指针运算与偏移 13](#_Toc29784)

[1.5.2 段错误 13](#_Toc8701)

[二. 条件结构与循环控制 14](#_Toc23367)

[三. 函数 15](#_Toc21997)

[3.1 声明与定义 15](#_Toc27360)

[3.1.1 声明函数原型 15](#_Toc29701)

[四. I/O(输入输出) 16](#_Toc15499)

[2.1 Unix I/O 16](#_Toc17225)

[2.1.1 打开与关闭文件 16](#_Toc10640)

[2.1.2 读与写文件 17](#_Toc29931)

[2.2 标准库文件IO 17](#_Toc3896)

[五. 网络编程 19](#_Toc23805)

[3.1 进程 19](#_Toc31101)

[3.1.1 创建进程 19](#_Toc1326)

[六. C++ 20](#_Toc27565)

[八. QT编程 21](#_Toc2998)

# 前言

* 在C++中引入标准C语言库函数时, 将 .h去掉, 在名称前加c.
* std 命名空间包含了C++标准库中的所有功能, 建议在大型项目中使用 std:: 前缀 而不是在整个文件使用 \*\*using namespace std\*\*. 这样可以避免潜在的命名冲突.

# 工程项目示例

|  |
| --- |
| project**/**  ├── src**/**  │ ├── main**.**cpp # 主程序文件**.**  │ ├── utils**/** # 通用工具函数**.**  │ │ ├── file\_utils**.**cpp  │ ├── modules**/** # 功能模块**.**  │ │ ├── module1**/**  │ │ │ ├── module1**.**cpp  │ │ │ └── module1**.**h  │ └── tests**/** # 测试代码**.**  │ ├── unit**/**  │ │ ├── test\_module1**.**cpp  │ └── integration**/**  │ ├── integration\_test1**.**cpp  ├── include**/** # 存放公共头文件**.**  │ ├── common**.**h  │ ├── utils**/**  │ │ ├── file\_utils**.**h  │ ├── modules**/**  │ │ ├── module1**.**h  ├── lib**/** # 存放第三方库文件**.**  │ ├── lib1**.**a  │ ├── lib2**.**so  ├── bin**/** # **(**与build类似**).**  ├── build**/** # 存放编译生成的目标文件和可执行文件**.**  ├── doc**/** # 存放项目文档**.**  │ ├── design**/**  │ │ ├── architecture**.**md  │ ├── api**/**  │ │ ├── api**.**md  └── Makefile # 构建项目的Makefile文件**.** |

# STL(Standard Template Libray)

* 容器是用于存储相同类型的数据的类模板, 按照数据结构可分为顺序容器和关联容器.
* 顺序容器: 存放多个单一类型的数据, 数据的存放位置仅仅与存入顺序有关. C++中自带的vector 和 list 都是类模板实现的.
* 关联容器: 数据存放的位置只和数据的大小有关, 与数据存入的先后顺序无关. 关联容器的底层都是通过二叉树来实现的.
  1. 顺序容器
     1. Vector
     2. List
     3. String
  2. 关联容器

1. 迭代器 iterator
2. 迭代器是一种对象, 能够用来遍历标准模板库容器中的部分或全部元素, 每个迭代器对象代表容器中元素的确定地址.
3. 迭代器提供了一些类似于指针的运算符: \* ++ == != =， 这些操作和c/c++操作指针的借口一致.
4. 迭代器虽然不是指针, 但是可以看做指向容器中元素的指针.

# 函数

## 2.1 函数模板

1. 在编译阶段, 编译器会根据函数模板的使用情况创建出函数名相同, 参数类型由编译器判断的若干函数. 通过函数模板创建的函数拥有相同的函数体, 只是函数的参数类型不同.
2. 每当在一个编译单元(经过预处理的.cpp文件)中使用了函数模板, 则必须在该编译单元中给出函数模板的定义. 因此建议在头文件中对函数模板进行声明定义.

|  |
| --- |
| // 声明(T 表示任意类型, 参数的类型和返回类型都可以指定为T.)  template**<**typename T**>**  返回类型 函数名称**(**参数列表**);**  // 定义(可以指定多个泛型类型)  template**<**typename T**,** typename U**>**  返回类型 函数名称**(**参数列表**)** **{**函数体**};** |

1. 特定的领域中工作时, 假设需要频繁使用一种特定类型的动态数组, 我们会考虑定义类型别名.

|  |
| --- |
| #include <vector>  template **<**typename T**>**  **using** data\_list **=** std**::**vector**<**T**>;**  data\_list |

### 2.1.1 函数模板特化

1. 在实例化函数模板时, 对特定类型的实参进行特殊的处理, 即当实参为特定类型时, 通过函数模板生成的函数会有不同的函数体.

|  |
| --- |
| template**<>**  返回类型 函数名称**<**特定的参数类型**>(**参数列表**)** **{**函数体**};** |

## 2.2 重载函数

1. 在源代码编写阶段, 所展示的函数名是一样的. 接下来进入到编译阶段, g++编译器会根据内部设定好的命名规则, 为重载函数重新命名. 其实到了汇编语言这一步, 重载函数命名已经有了区别. 最终汇编阶段汇编成二进制文件时, 不会发生编译错误.
2. 实际调用过程中, 如果实际调用传入的实参不符合任何一个重载函数, 会提升匹配: 通过数据类型兼容的形式, 将与形参类型不匹配的实参提升到能够匹配的程度. 例如: int与float能够相互兼容, 能够互相提升匹配.

* 如果经过一次提升匹配后, 发现有多个重载函数满足匹配, 将产生一个多重匹配的编译错误. 在设计函数的过程当中, 尽量避免出现多重匹配. 解决方式: ① 重载函数参数数量不同. ② 数据类型之间尽可能多的不兼容.

## 2.3 内联函数

1. 当该函数逻辑简单(通常少于5行),占用空间少并且被被频繁调用时, 通常的做法是省略原型直接将该函数定义在本应该提供原型的地方. 并且如果代码执行的时间很短, 内联函数调用可以节省相对于普通函数调用的大部分时间.

### 2.3.1 普通函数

1. 普通函数被调用时发生控制转移, 控制权(向内核申请运行资源)从主函数转移到被调用函数. 即普通函数发生调用后, 运行的所有资源(包括内存)都是需要向内核额外申请的. 此时被调用的函数拥有独立的内存空间.
2. 执行到普通函数的调用指令时, 程序将在函数调用后立即存储该指令的内存地址, 并将函数参数复制到堆栈, 跳到标记函数起点的内存单元, 执行函数代码(也许还需将返回值放到寄存器中), 然后跳回到地址被保存的指令处.

### 2.3.2 内联函数

1. 内联函数被调用时, 内联函数的定义部分会被复制粘贴到函数调用处的后面. 内联函数运行的时候占用主函数的空间, 如果内联函数中申请大量空间或逻辑复杂时对主函数的空间管理都有影响. 主函数向内核首先申请一个额定大小的空间, 运行过程当中如果不够再去申请.

|  |
| --- |
| /\*  功能: 使用内联函数计算参数的平方.  要点:  a. 内联函数中不允许使用循环语句, switch语句, 递归调用.  b. 为函数加上inline关键字后只是向编译器建议这个函数应该内联, 编译器会根据该函数的逻辑以及占用空间的大小决定是否采用这个建议.  c. 定义在类声明内部的方法都会被默认的建议成内联(隐式内联), 定义在类声明外部的方法称作显示内联函数.  \*/  #include <iostream>  #include <cmath>  **using** **namespace** std**;**  inline double square**(**double x**)** **{** **return** x **\*** x**;** **}**  int main**()** **{**  double a**,** b**,** c **=** 3.0**;**  a **=** square**(**1.0**);**  b **=** square**(**2.0 **+** 2.0**);**  cout **<<** "a = " **<<** a **<<** ", b = " **<<** b  **<<** ", c squared = " **<<** square**(**c**++)**  **<<** ", Now c = " **<<** c **<<** "\n"**;**  **return** 0**;**  **}**  注**:** 内联与宏 |

# 面向对象

## 3.1 类对象与成员初始化

* 类中的变量(任意类型)称为类的属性/成员变量, 类中的函数称为类的方法/成员函数.
* 类成员函数的定义一般写在与头文件对应的.cpp文件中, 并且当类的成员函数定义在类的外部时, 一定要在函数名称前面加上类名::, ::代表作用域运算符.

|  |
| --- |
| /\*  功能: 阐明类的不同形式与其对象初始化.  要点:  1. (推荐)公共类和其构造函数声明在头文件中.  2. 类在创建对象时可以省略struct/class.  3. class类只能通过构造函数来实现对象初始化, 而struct类还可以通过 "= {}" 实现对象初始化.  4. 类名(实参) 的初始化方式一般出现在等号右侧或者实参部分, 是一个临时对象. 例如可以写成 Data d = Data(a).  \*/  // struct 类  struct Example **{**  int a**;**  char b**;**  **};**  int main**()** **{**  Example example **=** **{**1**,** 'a'**};**  **return** 0**;**  **}**  // class 类  class Example **{**  int a**;**  char b**;**  public**:**  // 声明无参构造函数  Example**();**  // 声明一个参数的带参构造函数(explicit: 禁止隐式调用)  /\*explicit\*/ Example **(**int m**);**  // 声明两个参数的带参构造函数  Example **(**int m**,** char n**);**  **};**  // 定义无参构造函数进行初始化  Example**::**Example**()** **{**  a **=** 1**;**  b **=** 'a'**;**  std**::**cout **<<** "a = " **<<** a **<<** ", b = " **<<** b **<<** std**::**endl**;**  **}**  // 定义一个参数的带参构造函数进行初始化  Example**::**Example**(**int \_a**)** **{**  a **=** \_a**;**  std**::**cout **<<** a **<<** std**::**endl**;**  **}**  // 定义两个参数的带参构造函数进行初始化  Example**::**Example**(**int \_a， char \_b**)** **{**  a **=** \_a**;**  b **=** \_b**;**  std**::**cout **<<** "a = " **<<** a **<<** ", b = " **<<** b **<<** std**::**endl**;**  **}**  int main**()** **{**  // 无参构造函数初始化  Example example**;**  // 一个参数的带参构造函数初始化(隐式调用)  Example example **=** 1**;**  // 两个参数的带参构造函数初始化  Example example**(**1**,** 'a'**);**  **return** 0**;**  **}** |

* 类的访问限制符分为公开(public), 私有(private), 受保护的(protected)

|  |
| --- |
| /\*  功能: 区分三种访问限制符的使用方法和场景.  要点:  1. 未使用访问限制符时, class类的成员函数和成员变量默认是私有(private).  2. 类的外部, 对象可以通过运算符"."直接访问公有(public)成员. 如果是指针形式, 使用"->"访问.  3. 一般来说, 类中的所有成员变量(属性)写成私有, 所有函数(方法)写成公有, 少部分写成受保护的.  4. 私有成员如果需要在类外部进行操作, 需要设置对外操作接口.  \*/  class Example **{**  int a**;**  private**:**  int b**;**  public**:**  // 返回随机值(成员变量a未初始化)  int public\_function**()** **{**  int c **=** a**;**  **return** c**;**  **};**  // 设置对外访问接口  void setA**(**int \_a**)** **{** a **=** \_a**;** **}**  int getA**()** **{** **return** a**;** **}**  protected**:**  int d**;**  **};**  int main**()** **{**  // 成员变量a,b,d均不允许在外部访问.  Example example**;**  Example**\*** ptr **=** **&**example**;**  std**::**cout **<<** example**.**public\_function**()** **<<** std**::**endl**;**  std**::**cout **<<** ptr **->** public\_function**()** **<<** std**::**endl**;**  // 类外部修改和访问私有成员变量a的值.  example**.**setA**(**1**);**  std**::**cout **<<** example**.**getA**()** **<<** std**::**endl**;**  return0**;**  **}** |

## 3.2 构造函数

构造函数是创建对象时同步调用的特殊成员函数, 用于初始化对象的状态, 其推荐定义在类的外部.

构造函数需要满足三个条件: ① 在类中进行声明. ② 没有返回值. ③函数名和类名一致.

如果类当中的一部分数据是声明在堆空间上的, 析构函数需要手动的释放堆空间. 手动调用会造成内存错误.

|  |
| --- |
| /\*  功能: 举例不同形式的构造函数.  要点:  a. 类没有声明任何构造函数时, 编译器会生成一个无参无内容的构造函数---默认构造函数(类名(){};).  b. 当类构建对象的时候会构建所有的属性.  c. 类方法会使用一个隐藏的参数(this指针)接受类对象调用方法时默认传入的对象的地址. 即所有的类的方法中, 隐藏了一个 this指针(指针常量), 指向了调用该方法的对象的地址. 例如调用example.show(), 编译器认为的调用的是example.show(&example).  d. 引用(type& refName = varName): ① 引用必须在定义的同时进行初始化且不能为NULL(常量只能被常量引用绑定); ② 引用绑定成功之后不能更改绑定; ③ 函数的形参直接设置为引用, 则函数内部操作形参相当于在操作实参本身; ④ 引用在内存中不占据空间, sizeof引用返回引用所绑定的变量的大小.  \*/  // 带参构造函数(接受一个或多个参数, 并且可以被重载)  class Example **{**  private**:**  int a**;**  int b**;**  public**:**  // 使用this进行常规写法的初始化时, 参数名可以和成员变量名相同.  Example**(**int a**,** int b**)** **{**  **this->**a **=** a**;**  **this->**b **=** b**;**  std**::**cout **<<** "normal writing: a = " **<<** a **<<** ", b = " **<<** b **<<** std**::**endl**;**  **}**    // 普通成员的列表初始化中成员的顺序必须和类成员声明的顺序一致  Example**(**int a**,** int b**):**a**(**a**),** b**(**b**){**  std**::**cout **<<** "list initialization: a = " **<<** a **<<** ", b = " **<<** b **<<** std**::**endl**;**  **}**  **};**  class Test **{**  private**:**  Example example**;**  public**:**  // 在当前构造函数中对其它类对象的构造函数进行列表初始化  Test**():**example**(**1**,** 2**){}**  **};**  int main**(**int argc**,** char**\*\*** argv**)** **{**  Test test**;**  **return** 0**;**  **}**  // 拷贝构造函数 |

友元

友元函数

友元类

当一个类A里面将外部函数func声明成一个友元函数的话, 则func允许直接访问类A里面的私有成员.

一个函数或另一个类中的成员函数可以访问类中的非共有成员.

friend 函数返回类型 函数名(参数列表)

运算符重载函数

引用

为了尽量少的使用指针, 创建了数据类型: \*\*引用\*\*. 对\*\*引用\*\*进行的所有操作都是在与之绑定的对象上进行, 它们的类型必须相同, 而且绑定后无法解除.

作为函数参数, 可以通过引用传递参数, 以便在函数内部修改原始变量的值.

在函数返回时，可以使用引用返回多个值

用于创建别名，可以简化变量的操作

|  |
| --- |
| /\*  功能: 利用拷贝构造函数示例说明引用的应用场景.  要点:  a. 引用的编码格式: 等待绑定的变量的数据类型& 引用名称 = 等待绑定的变量;    \*/  // main.cpp  #include  int**&** a\_ **=** a**;** |

new与delete

与 C 语言申请堆内存不同, new 与 delete 在为对象申请和释放堆内存时会调用构造函数和析构函数.

| 写法 | 描述 |

| :------------------------------------: | :----------------------------------: |

| 类名\* 变量名 = new 构造函数(参数列表); | 创建对象 |

| 类名\* 变量名 = new 类名[数组大小]; | 创建对象数组, 此时会调用多次构造函数 |

| delete 指向对象的指针; | 销毁对象, 会调用析构函数 |

| delete []对象数组名; | 销毁对象数组, 此时会调用多次析构函数 |

## 

## 继承

## public 受保护成员继承过后可以内部访问.

## protected 私有成员被隐藏, 其它成员变成受保护成员

## private继承: 私有成员被隐藏, 其它成员变成私有成员.

## 基类的构造函数是在派生类的构造函数的列表初始化的地方被调用的。我们不去写，默认调用

## 基类的无参的构造函数，

## 当派生类和基类拥有一样的属性时, 派生类直接访问该属性，访问的一定是派生类自己的。如果想要访问基类的方法, 应该调用基类的公开方法.

## 类中的成员方法重名(称作函数重写/覆盖)问题: 要求函数名必须相同, 形参列表和返回值无所谓（仅仅在继承过程当中如此）

## 被覆盖的函数还是可以被访问到，不是真正的覆盖： 直接在方法前面加上 基类的作用域::

## 步骤

1. 预处理(Preprocessing): 预处理器会执行一系列的预处理指令(以#开头), 如宏替换, 宏展开, 文件包含, 删除注释等. 预处理后的代码会被输出到一个临时文件中, 通常以`.i`, `.ii`, `.cpp`作为扩展名.
2. 编译阶段(Compilation): 编译器(gcc, g++等)会将预处理后的代码转换为汇编语言代码. 这个过程涉及词法分析, 语法分析, 语义分析等步骤. 编译后的汇编代码会被保存在一个以`.s`为扩展名的文件中.
3. 汇编阶段(Assembly): 汇编器(as, gas等)会将汇编语言代码转换成机器指令, 生成一个二进制的目标文件(通常以`.o`或`.obj`作为扩展名). 其中包含了机器指令, 符号表等信息.
4. 链接阶段(Linking): 链接器(ld, g++等)将多个目标文件以及可能的库文件合并成一个可执行程序. 这个过程涉及解析外部引用, 解决符号重定位等步骤, 最终生成一个可以在操作系统上运行的可执行文件.

* 静态库在编译时与程序链接, 链接器将库的代码和程序复制到可执行文件中. 静态库文件较大, 但是在程序运行时不需要外部库文件, 具有更好的移植性. Windows和类Unix操作系统中静态库文件的标准命名约定分别是以.lib与.a后缀的文件.
* 共享库在程序运行时与程序动态链接, 共享库的大小相对较小, 共享库在程序运行时需要确保库文件存在于系统中. Windows和类Unix操作系统中共享库文件分别是使用.dll(Dynamic Link Library)和.so(Shared Object)后缀的文件.

## 示例

|  |
| --- |
| /\*  文件名: for\_calculation.c  功能: 通过循环计算的示例代码说明编译器gcc的用法.  \*/  #include <stdio.h>  int main**(**void**)** **{**  double counter**;**  double result**;**  double temp**;**  **for** **(**counter **=** 0**;** counter **<** 4000.0 **\*** 4000.0 **\*** 4000.0 **/** 20.0 **+** 2030**;**counter **+=** 5**/**4**)** **{**  temp **=** counter **/** 1239**;**  result **=** counter**;**  **}**  printf**(**"Result is %lf\n"**,** result**);**  **return** 0**;**  **}**  // 进行预编译  gcc **-**E for\_calculation**.**c **-**o for\_calculation**.**i  // 编译生成汇编(assembly)代码  gcc **-**S for\_calculation**.**i **-**o for\_calculation**.**s  // 汇编代码转换成二进制目标文件  gcc **-**c for\_calculation**.**s **-**o demo**.**o  /\*  a. -O 减少代码的长度与执行时间, 效果等价于-O1, 其中包括线程跳转和延迟退栈.  b. -O2 完成所有-O1级别的优化之外, 同时进行额外调整工作(处理器指令调度等).  c. -O3 完成所有-O2级别的优化之外, 包括循环展开和其它与处理器特性相关的优化工作.  d. 数字越大优化的等级越高, 意味着程序运行越快. 一般选用-O2选项, 它在优化长度, 编译时间和代码大小之间取得了比较理想的平衡点.  \*/  // 指定语言标准并以-O2选项优化  gcc **--**std**=**c99 **-**O2 for\_calculation**.**c **-**o for\_calculation |

# 数据类型

## 1.1 进程内存分配

1. 栈区(stack)
2. 主要存储函数调用的局部变量, 函数参数以及函数调用的上下文信息.
3. 在栈上(先进后出的数据结构)分配的数据量有限并且其内存的分配和释放都是自动的, 过多的栈区内存使用可能导致栈溢出.
4. (运行时分配)运行到变量定义语句才分配, 分配失败程序崩溃.
5. 堆区(heap)
6. 刚分配出来的内存块中的内容是野值, 不可以直接使用.
7. 不使用的内存块及时释放, 否则可能会造成内存泄漏. 但是同一内存块不可以重复释放, 否则会引起程序崩溃.
8. (运行时分配)分配和释放通过函数(如`malloc`和`free`)或运算符(如`new`和`delete`)来实现.
9. 数据区
10. 静态存储区: 全局变量存储在静态存储区, 使用 `static`修饰的全局变量生命周期不变而作用域限制在当前文件, 使用 `static`修饰的局部变量生命周期加长到程序结束而作用域不变, 其内存地址在编译时就已确定.
11. 代码区

## 1.2 变量的存储类型

1. auto(自动变量)

局部变量在缺省存储类型的情况下归为自动变量, 默认不会初始化并产生随机值.

1. static(静态变量)

(分为全局变量和局部)默认会初始化并仅初始化一次.

1. extern(外部变量(全局变量))

作用域为整个程序(包含该程序的各个文件). 默认会初始化并仅初始化一次. 尝试使用外部全局变量, 引用失败则创建一个新的全局变量.

1. register(寄存器变量)

只能是局部变量不能是全局变量并且不能加 static 修饰, 默认不会初始化.

注意:

1. 类似于 extern int a; 被称为引用性声明(未分配内存空间), 可以进行重复声明.
2. 类似于 int a; 严格意义上被称为定义性声明(分配内存空间). 类似于 int a = 0; 被称为定义.
3. 如果全局变量或静态变量没有显示初始化, 则会被默认初始化为相应类型的零值. 局部变量(分配在栈上的变量)的默认初始值是不确定的, 最好显示对其进行初始化. 指针变量的默认初始化是指初始化为`NULL`.
4. 对于循环次数较多的循环控制变量及循环体内反复使用的变量均可定义为寄存器变量.

## 1.3 基本数据类型

### 1.3.1 整型

|  |
| --- |
| /\*  功能: 示例整型数据如何定义.  要点:  a. 整型数据在内存中的存放形式是以补码表示的. 我们使用32个二进制位(数字电路高电平1, 低电平0)表示整型的正数和负数补码. 其中, 正数的补码与原码相同, 而负数的补码是除第一位符号位外, 其余位全部求反后加1.  b. 后缀 'l'/'L', 'u'/'U' 分别表示长整型和无符号数.  c. sizeof(int)返回的是 size\_t类型(无符号整数类型).  \*/  int a **=** 10**;** **(**原码**:** 000...1010 补码**:** 000...1010**)**  int a **=** **-**10**;** **(**原码**:** 100...1010 补码**:** 111...0110**)**  // 下述相关数据仅限于32位系统(6种整型类型的容纳范围可查看 <limits.h>).  int main**()** **{**  /\* [signed] int(整型), 32位, -2^32至2^32-1 \*/  int a **=** 0**;**  printf**(**"Number of memory bytes occupied: %lu\n"**,** **sizeof(**int**));**    /\* unsigned int(无符号整型), 32位, 0至2^32-1 \*/  unsigned b **=** 0U**;**  printf**(**"Number of memory bytes occupied: %lu\n"**,** **sizeof(**unsigned int**));**    /\* [signed] short int(短整型), 16位, -32768至32767 \*/  short c **=** 0**;**  printf**(**"Number of memory bytes occupied: %lu\n"**,** **sizeof(**short int**));**    /\* unsigned short int(无符号短整型), 16位, 0至65535 \*/  unsigned short d **=** 0U**;**  printf**(**"Number of memory bytes occupied: %lu\n"**,** **sizeof(**unsigned short int**));**    /\* [signed] long int(长整型), 32位, -2147483648至2147483647 \*/  long e **=** 057L**(**八进制长整型**);**  printf**(**"Number of memory bytes occupied: %lu\n"**,** **sizeof(**long int**));**    /\* unsigned long int(无符号长整型), 32位, 0至4294967295 \*/  unsigned long f **=** 0X25ABUL**(**十六进制无符号长整形**);**  printf**(**"Number of memory bytes occupied: %lu\n"**,** **sizeof(**unsigned long int**));**  **}** |

### 1.3.2 字符型

|  |
| --- |
| /\*  功能: 字符型数据相关操作  要点: a. 字符是指字符集中的任意字符, 字符变量的值以ASCII码形式存放在内存单元之中.  b. 转义字符是一种特殊的字符常量, 常见的转义字符有 \n(回车换行), \r(回车)等. C语言字符集中任何一个字符均可用转义字符来表示.  c. 字符常量占一个字节的内存空间, 字符串常量占的内存字节数等于字符串中字节数加1.  d. 每个字符变量被分配一个字节的内存空间(只存放一个字符), 字符值以ASCII码形式存放在变量的内存单元之中.  \*/  int main**()** **{**  // [signed] char(有符号字符型), -128至127,  // unsigned char(无符号字符型), 0至255  **}** |

### 1.3.3 浮点型(实型)

1. 单精度型: 类型声明符为 float, 在内存中占四个字节(32位).
2. 双精度型: 类型声明符为 double, 在内存中占八个字节(64位).
3. 长双精度型: 类型声明符为 long double, 在内存中占十六个字节(128位).

## 1.4 构造数据类型

### 1.4.1 数组类型

1. 字符初始化字符数组时, 建议最后一个字符以结束符\0结尾. 字符串初始化字符数组时多占用一个字节(结束符\0), 并且一般无需指定数组长度.
2. 数组名是整个数组的首地址, 它是一个常量不允许直接赋值. 整个数组存储在一块连续的内存单元中, 对首地址加上偏移量可以指向其它元素的存储单元.
3. 在C语言中, 访问数组的越界索引是未定义行为. 这意味着编译器不会对此进行错误检测, 而是允许代码继续执行. 编译器可能会将越界访问的结果视为随机值, 崩溃程序或产生其它意外结果.

### 1.4.2 结构体类型

1. 结构体的组成部分都存放在内存中一段连续的区域内, 而指向结构体的指针就是结构体第一个字节的地址. 编译器维护关于每个结构体类型的信息, 并指示每个字段(field)的字节偏移. 它以这些偏移作为内存引用指令中的位移, 从而产生对结构体元素的引用.
2. 声明结构体类型(头文件中)时系统不对其分配实际内存, 只有定义相应的变量才实际分配内存.
3. 结构体变量的赋值是给结构体变量的各成员进行赋值, 并且允许具有相同类型的结构体变量相互赋值. 程序中可以引用结构体变量成员的地址, 也可以引用结构体变量的地址.

### 1.4.3 枚举类型

|  |  |
| --- | --- |
| /\*  功能: 枚举数据相关操作.  要点:  a. 声明(不分配内存空间):  enum 枚举类型名 {枚举值...};  b. 定义:  enum 枚举类型名 枚举变量...;  enum 枚举类型名 {枚举值...} 枚举变量列表;  enum {枚举值...} 枚举变量列表;  c. 枚举类型是符号常量不能使用赋值语句, 枚举变量可以通过枚举值对其进行赋值.  \*/  // 声明枚举类型可同时对枚举值赋值.  enum Car\_TYpe **{**BENZ**=**2**,** BWM**,** FORD**=**8**,**AUDI**,** BYD**};**  enum Car\_Type my\_car**=**BENZ**,** YOUR\_CAR**=**BYD**,** other\_car**=**9 // other\_car 不建议直接赋值.  // 赋值 BENZ=0 出错, 不能对枚举值(符号常量)进行赋值.  enum Car\_Type **{**BENZ**,** BWM**,** BYD**};**  enum Car\_Type my\_car **=** BENZ**,** your\_car**;**  enum Car\_Type **{**BENZ**,** BWM**,** BYD**}** my\_car**=**BENZ**,** your\_car**;**  enum **{**BENZ**,** BWM**,** BYD**}** my\_car**=**BENZ**,** your\_car**;** |  |

### 1.4.4 共用体(联合类型)

## 1.5 指针类型

1. 内存以字节为单位进行编址, 其中每个字节都对应一个地址. 变量对应内存中的一个存储单元, 用存储单元的第一个字节的地址表示变量的地址.
2. 通常32位操作系统指针变量占用内存大小是4字节, 64位操作系统是8字节.
3. 指针定义后如果不进行初始化会产生野指针, 当初始化为NULL时为空指针, 即值为0, 不指向任何变量.
4. 常量指针(const 类型说明符\* 指针变量)通常作为函数形参使用, 目的是在函数中不允许通过该常量指针去修改它所指向的存储单元的值. 可以通过新的指针去修改.
5. 指针常量(类型说明符\* const 指针变量)本身的地址值不可改变, 可以通过二级指针修改.

### 1.5.1 指针运算与偏移

### 1.5.2 段错误

1. 通常段错误就是指: 访问的内存超出了系统所给的这个程序的内存空间. 一般出现在运行阶段而不是编译阶段.
2. 引起段错误的操作
3. 操作空指针.
4. 操作地址不存在的内存.
5. 操作受系统保护的内存.
6. 修改常量区的内容.
7. 解决建议
8. 变量 / 数组 / 指针 均要初始化, 最好是一个具体值.
9. 数组不要越界操作(不要随意去操作不存在或受系统保护的内存).

# 条件结构与循环控制

# 函数

## 3.1 声明与定义

### 3.1.1 声明函数原型

若在调用函数之后才对被调用的函数进行定义, 则需要在调用函数之前声明, 否则会出现编译器警告隐式声明(C标准已不推荐).

|  |
| --- |
| /\*  功能: 两种不同的函数原型声明(可放在公共头文件).  要点:  1. 请勿重复定义函数: 返回类型 函数名称(类型 形参...) { 函数体 };  \*/  #include <stdio.h>  // 类型说明符 函数名称(类型...);  int getMembershipInfo**(**int**);**  // 类型说明符 函数名称(类型 形参...);  int getMembershipInfo**(**int userId**);**  int main**()** **{**  int rest\_days**;**  rest\_days **=** getMembershipInfo**(**1**);**  **return** rest\_days**;**  **}**  int getMembershipInfo**(**int userId**)** **{**  int fetchData **=** 100**;**  **return** fetchData**;**  **}** |

# I/O(输入输出)

## 2.1 Unix I/O

1. 所有的I/O设备(网络, 磁盘, 终端等)都被模型化为文件, 所有的输入输出都被当做对相应文件的读和写来执行. 这种将设备优雅映射为文件的方式, 允许Linux内核引出一个简单且低级的应用接口, 称为Unix I/O.

### 2.1.1 打开与关闭文件

1. 应用程序要求内核打开相应的文件, 内核返回一个描述符(进程中当前没有打开的最小非负整数). 应用程序需要记住这个描述符, 而内核会记录有关这个打开文件的所有信息.
2. 应用程序完成了对文件的访问之后会通知内核关闭文件. 内核会释放文件打开时创建的数据结构, 并将这个描述符恢复到可用的描述符池中.

|  |
| --- |
| /\*  功能: 创建一个新文件, 设置文件所有者有读写权限, 其他用户只有读权限,  要点:  a. open函数将filename转换为一个文件描述符, 并且返回描述符数字.  b. flags:  O\_RDONLY 只读  O\_WRONLY 只写  O\_RDWR 可读可写  O\_CREAT 文件不存在则创建一个截断的空文件  O\_TRUNC 文件若存在则截断  O\_APPEND 设置文件位置到文件末尾处  c. 进程通过带某个mode参数的 open函数调用来创建一个新文件时, 文件的访问权限位(定义在sys/stat.h中)被设置为 mode & ~ umask.  d. Linux/Unix中, sudo su 用来切换到root用户而不需要密码. su [用户名] 用来完全切换到其他用户.  \*/  // 打开文件成功返回新文件描述符, 出错返回-1.  int open**(**char**\*** filename**,** int flags**,** mode\_t mode**);**  // 关闭文件成功返回0, 出错返回-1.  int close**(**int fd**);**  // 功能示例代码  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>  #define DEF\_MODE S\_IRUSR|S\_IWUSR|S\_IRGRP|S\_IWGRP|S\_IROTH|S\_IWOTH  #define DEF\_UMASK S\_IWGRP|S\_IWOTH  int main**()** **{**  /\* 设置屏蔽位 \*/  umask**(**DEF\_UMASK**);**  int fd **=** open**(**"demo.txt"**,** O\_CREAT**|**O\_TRUNC**|**O\_WRONLY**,** DEF\_MODE**);**  close**(**fd**);**  **return** 0**;**  **}** |

### 2.1.2 读与写文件

## 2.2 标准库文件IO

1. 从文件编码方式分为文本文件和二进制文件.
2. 文本文件: 一个字节存放一个ASCII字符. 占用存储空间较多, 转换为二进制速度慢.
3. 二进制文件: 以补码形式存放. 一个字节不对应一个字符(不能直接输出字符形式). 占用存储空间较少.
4. 目前C语言所使用的的磁盘文件系统分为缓冲文件系统和非缓冲文件系统.
5. 缓冲文件系统: 缓冲文件系统自动地在内存区为每一个正在使用的文件开辟文件信息区. 从磁盘向内存读入数据时, 则一次从磁盘文件将一些数据输入到内存缓冲区(充满缓冲区), 然后再从缓冲区逐个将数据送给接收变量. 如果需要访问相同的数据, 直接从缓冲区中直接获取, 不需要从磁盘读取. 而向磁盘文件写入数据时, 先将数据送到内存中的缓冲区, 装满缓冲区后才一起送到磁盘去. 通过在内存中维护缓冲区, 减少对磁盘的实际读写操作, 提高文件读写性能.
6. 非缓冲文件系统: 通常用于需要直接控制数据传输的高性能应用, 由用户自己根据需要设置.
7. 在传统的UNIX系统中, 用缓冲文件系统来处理文本文件, 用非缓冲文件系统处理二进制文件. 但是1983年ANSI C标准决定不采用非缓冲文件系统, 而只采用缓冲文件系统, 即将缓冲文件系统扩充为可以处理二进制文件.
8. 一般把缓冲文件系统的输入输出称为标准输入输出(标准I/O), 非缓冲文件系统的输入输出称为系统输入输出(系统I/O).

- 流根据数据内容可分为文本流和二进制流.

- \*\*文本流\*\*: 流动的字符序列.

- \*\*二进制流\*\*: 流动的二进制序列.

# 网络编程

## 3.1 进程

1. Linux shell 创建的每个进程开始时都有三个打开的文件: 标准输入(描述符0), 标准输出(描述符1), 标准错误(描述符2).
2. 进程控制块(Process Control Block, 也称进程表项)是存放进程管理和控制信息的task\_struct(结构体). 其中存放了进程标识符, 处理器状态信息, 进程调度信息, 进程控制信息.
3. 处理器运行时的各种状态信息(通用寄存器值, 程序计数器值, 堆栈指针, 标志寄存器等)存放在寄存器中, 当被中断时这些信息要存放在PCB里.
4. 进程创建过程: 申请空白PCB->为新进程分配资源->初始化PCB->插入就绪队列
5. ps命令中STAT表示进程的状态: 静止/阻塞态(S), 暂停/就绪态(T), 运行态(R), Zombie State(Z, 僵尸态), +表示位于后台的进程.

### 3.1.1 创建进程

3.1.2 僵尸进程

3.1.3 孤儿进程

3.1.4 守护进程

# C++

# QT编程

4.1 环境搭建

1. Qt基础模块: Qt core / Qt Gui / Qt Multimedia / Qt Network / Qt Qml / \*\*Qt Quick\*\* / \*\*Qt SQL\*\* / \*\*Qt Test\*\* / \*\*Qt Webkit\*\*.

- Qt扩展模块: \*\*Qt 3D\*\* / \*\*Qt Bluetooth\*\* / \*\*Qt Contacts\*\* / \*\*Qt Concurrent\*\* / \*\*Qt D-Bus\*\* / \*\*Qt OpenGL\*\* / \*\*Qt Location\*\*等.

> Qt Location 提供定位, 地图, 导航, 位置搜索等功能的QML和C++ API.

#### Qt开发工具(qttools)

- Qt Creator: 轻量级跨平台集成开发工具(IDE). 提供了代码编辑/调试/编译/部署等功能.

- Qt Designer: 强大的拖拽式图形化用户界面排版和设计工具.

- Qmake: 跨平台Makefile生成器.

> Qt 基于编程语言C++, 所以Qt 源文件依旧是 .cpp 后缀, 编译器依旧是 g++. 但是我们直接使用 g++ 去编译 Qt 的程序时无法编译, 我们必须使用 Makefile 文件去编译. Makefile 会针对当前目录下的所有文件进行编译.

- Qt Assistant: 可定制可重新发布的帮助文档和文档阅读器.

- Qt Linguist: 多语言支持工具，用于管理和翻译 Qt 应用程序的界面文本.

### 1.2 Linux 可视化安装

- 这里提供Qt相关地址: https://download.qt.io/.

4.1 QT 程序编译流程

1. qmake -project: 根据当前Qt程序创建工程文件, 该工程文件说明了当前程序包含的头文件.
2. qmake: 根据工程文件自动生成Makefile文件.
3. make: 运行Makefile里面所有编译语句, 运行完毕会自动生成一个和目录名称相同的可执行文件.