**LẬP TRÌNH C**

[CHƯƠNG 1: BẮT ĐẦU VỚI NGÔN NGỮ C 1](#_Toc145149811)

[1. Hello World 1](#_Toc145149812)

[PHẦN 1.2 BẢN GỐC “HELLO WORLD!” TRONG K&R C 3](#_Toc145149813)

[CHƯƠNG 2: COMMENTS 6](#_Toc145149814)

[PHẦN 2.1 CHÚ THÍCH BẰNG CÁCH SỬ DỤNG TIỀN XỬ LÝ(PREPROCESSOR) 6](#_Toc145149815)

[PHẦN 2.2 CHÚ THÍCH ĐƯỢC GIỚI HẠN BẰNG /\* \*/ 6](#_Toc145149816)

[PHẦN 2.3 CHÚ THÍCH ĐƯỢC GIỚI HẠN BẰNG DẤU // 7](#_Toc145149817)

[PHẦN 2.4 RỦI RO CÓ THỂ XẢY RA DO TRIGRAPH 8](#_Toc145149818)

[CHƯƠNG 3: KIỂU DỮ LIỆU 9](#_Toc145149819)

[PHẦN 3.1 GIẢI THÍCH CÁC KHAI BÁO 9](#_Toc145149820)

[PHẦN 3.2 KIỂU SỐ NGUYÊN CÓ ĐỘ RỘNG CỐ ĐỊNH ( KỂ TỪ C99) 11](#_Toc145149821)

[PHẦN 3.3 KIỂU SỐ NGUYÊN VÀ HẰNG SỐ 12](#_Toc145149822)

[PHẦN 3.4 HẰNG SỐ THỰC 14](#_Toc145149823)

[PHẦN 3.5 CHUỖI KÍ TỰ 14](#_Toc145149824)

[CHƯƠNG 4: TOÁN TỬ 17](#_Toc145149825)

[PHẦN 4.1 TOÁN TỬ QUAN HỆ 17](#_Toc145149826)

[4.1.1 Bằng “==” 17](#_Toc145149827)

[4.1.2 Không bằng (Khác) “!=” 17](#_Toc145149828)

[4.1.3 Not “!” 18](#_Toc145149829)

[4.1.4 So sánh lớn hơn “>” 18](#_Toc145149830)

[4.1.5 So sánh bé hơn “<” 18](#_Toc145149831)

[4.1.6 So sánh lớn hơn hoặc bằng “>=” 18](#_Toc145149832)

[4.1.7 So sánh bé hơn hoặc bằng “<=” 18](#_Toc145149833)

[PHẦN 4.2 TOÁN TỬ ĐIỀU KIỆN/ TOÁN TỬ 3 NGÔI 18](#_Toc145149834)

[PHẦN 4.3 TOÁN TỬ BITWISE 20](#_Toc145149835)

[PHẦN 4.4 HOẠT ĐỘNG NGẮN MẠCH CỦA TOÁN TỬ LOGIC 22](#_Toc145149836)

[PHẦN 4.5 TOÁN TỬ DẤU PHẨY 22](#_Toc145149837)

[PHẦN 4.6 TOÁN TỬ SỐ HỌC 23](#_Toc145149838)

[4.6.1 Số học cơ bản 23](#_Toc145149839)

[4.6.2 Toán tử cộng 23](#_Toc145149840)

[4.6.3 Toán tử trừ 23](#_Toc145149841)

[4.6.4 Toán tử nhân 24](#_Toc145149842)

[4.6.5 Toán tử chia 24](#_Toc145149843)

[4.6.6 Toán tử modulo (Phép chia lấy dư) 26](#_Toc145149844)

[4.6.7 Toán tử tăng giảm 26](#_Toc145149845)

[PHẦN 4.7 TOÁN TỬ TRUY CẬP 27](#_Toc145149846)

[4.7.1 Thuộc tính của đối tượng 27](#_Toc145149847)

[4.7.2 Thành viên của đối tượng được trỏ tới 28](#_Toc145149848)

[4.7.3 Lấy địa chỉ 28](#_Toc145149849)

[4.7.4 Dereference 29](#_Toc145149850)

[4.7.5 Chỉ số mảng 29](#_Toc145149851)

[4.7.6 Khả năng hoán đổi của chỉ số mảng 29](#_Toc145149852)

[PHẦN 4.8 TOÁN TỬ SIZEOF 29](#_Toc145149853)

[4.8.1 With a type as operand 29](#_Toc145149854)

[4.8.2 With an expression as operand 30](#_Toc145149855)

[PHẦN 4.9 TOÁN TỬ ÉP KIỂU 30](#_Toc145149856)

[PHẦN 4.10 TOÁN TỬ GỌI HÀM 30](#_Toc145149857)

[PHẦN 4.11 TOÁN TỬ TĂNG/ TOÁN TỬ GIẢM 31](#_Toc145149858)

[PHẦN 4.12 TOÁN TỬ GÁN 31](#_Toc145149859)

[PHẦN 4.13 TOÁN TỬ LOGIC 33](#_Toc145149860)

[4.13.1 Logic AND 33](#_Toc145149861)

[4.13.2 Logic OR 33](#_Toc145149862)

[4.13.3 Logic NOT 34](#_Toc145149863)

[PHẦN 4.14 TOÁN HỌC TRÊN CON TRỎ 35](#_Toc145149864)

[4.14.1 Cộng con trỏ 35](#_Toc145149865)

[4.14.2 Trừ con trỏ 35](#_Toc145149866)

[PHẦN 4.15 \_Alignof 35](#_Toc145149867)

[CHƯƠNG 5: BOOLEAN 35](#_Toc145149868)

[PHẦN 5.1 SỬ DỤNG STDBOOL.H 35](#_Toc145149869)

[PHẦN 5.2 SỬ DỤNG #DEFINE 36](#_Toc145149870)

[PHẦN 5.3 SỬ DỤNG KIỂU \_BOOL NỘI TẠI(TÍCH HỢP) 37](#_Toc145149871)

[PHẦN 5.4 SỐ NGUYÊN VÀ CON TRỎ TRONG BIỂU THỨC BOOLEAN 38](#_Toc145149872)

[PHẦN 5.5 XÁC ĐỊNH KIỂU BOOL BẰNG CÁCH SỬ DỤNG TYPEDEF 39](#_Toc145149873)

[CHƯƠNG 6: CHUỖI 41](#_Toc145149874)

[PHẦN 6.1 PHÂN TÁCH TỪ: STRTOK(), STRTOK\_R() VÀ STRTOK\_S() 41](#_Toc145149875)

[PHẦN 6.2 STRING LITERALS 44](#_Toc145149876)

[PHẦN 6.3 TÍNH ĐỘ DÀI CHUỖI: STRLEN() 45](#_Toc145149877)

[PHẦN 6.4 GIỚI THIỆU CƠ BẢN VỀ CHUỖI 46](#_Toc145149878)

[PHẦN 6.5 SAO CHÉP CHUỖI 47](#_Toc145149879)

[6.5.1 Gán con trỏ không sao chép chuỗi 47](#_Toc145149880)

[6.5.2 Sao chép chuỗi dùng các hàm tiêu chuẩn 48](#_Toc145149881)

[6.5.2.1 strcpy() 48](#_Toc145149882)

[6.5.2.2 snprintf() 48](#_Toc145149883)

[6.5.2.3 strncat() 49](#_Toc145149884)

[6.5.2.4 strncpy() 50](#_Toc145149885)

[PHẦN 6.6 LẶP QUA CÁC KÍ TỰ TRONG MỘT CHUỖI 50](#_Toc145149886)

[PHẦN 6.7 TẠO MẢNG CHUỖI 51](#_Toc145149887)

[PHẦN 6.8 CHUYỂN ĐỔI CHUỖI THÀNH SỐ: ATOI(), ATOF() (CẨN THẬN KHÔNG NÊN SỬ DỤNG) 52](#_Toc145149888)

[PHẦN 6.9 ĐỌC GHI DỮ LIỆU DẠNG CHUỖI 53](#_Toc145149889)

[PHẦN 6.10 TÌM VỊ TRÍ XUẤT HIỆN ĐẦU TIÊN/ CUỐI CÙNG CỦA MỘT KÍ TỰ CỤ THỂ: STRCHR(), STRRCHR() 54](#_Toc145149890)

[PHẦN 6.11 SAO CHÉP VÀ NỐI CHUỖI: STRCPY(), STRCAT() 55](#_Toc145149891)

[PHẦN 6.12 SO SÁNH: STRCMP(), STRNCMP(), STRCASECMP(), STRNCASECMP() 56](#_Toc145149892)

[PHẦN 6.13 CHUYỂN ĐỔI CHUỖI THÀNH SỐ MỘT CÁCH AN TOÀN: HÀM STRTOX 59](#_Toc145149893)

[PHẦN 6.14 STRSPN VÀ STRCSPN 61](#_Toc145149894)

[CHƯƠNG 7: CHỮ SỐ, KÝ TỰ VÀ CHUỖI 62](#_Toc145149895)

[PHẦN 7.1 FLOATING POINT LITERALS( CHỮ SỐ DẤU CHẤM ĐỘNG) 62](#_Toc145149896)

[PHẦN 7.2 STRING LITERALS( CHUỖI KÍ TỰ CHỮ) 62](#_Toc145149897)

[PHẦN 7.3 CHARACTER LITERALS(KÝ TỰ) 63](#_Toc145149898)

[PHẦN 7.4 INTEGER LITERALS(SỐ NGUYÊN) 64](#_Toc145149899)

[CHƯƠNG 8: COMPOUND LITERALS 65](#_Toc145149900)

[PHẦN 8.1 ĐỊNH NGHĨA/ KHỞI TẠO COMPOUND LITERALS 65](#_Toc145149901)

[CHƯƠNG 9: BIT- FIENDS 67](#_Toc145149902)

[CHƯƠNG 10: MẢNG 69](#_Toc145149903)

[PHẦN 10.1 KHAI BÁO VÀ KHỞI TẠO MẢNG 69](#_Toc145149904)

[PHẦN 10.2 LẶP QUA MẢNG HIỆU QUẢ VÀ THỨ TỰ ƯU TIÊN HÀNG( ROW MAJOR ORDER) 70](#_Toc145149905)

[PHẦN 10.3 ĐỘ DÀI MẢNG 72](#_Toc145149906)

[PHẦN 10.4 TRUYỀN MẢNG ĐA CHIỀU VÀO HÀM 74](#_Toc145149907)

[PHẦN 10.5 MẢNG ĐA CHIỀU 75](#_Toc145149908)

[10.5.1 Mảng hai chiều 75](#_Toc145149909)

[10.5.2 Khởi tạo mảng 2 chiều 76](#_Toc145149910)

[10.5.3 Truy cập phần tử trong mảng hai chiều 76](#_Toc145149911)

[10.5.4 Mảng ba chiều 77](#_Toc145149912)

[10.5.5 Bộ nhớ mảng ba chiều 78](#_Toc145149913)

[10.5.6 Khởi tạo một mảng ba chiều 78](#_Toc145149914)

[PHẦN 10.6 DEFINE MẢNG VÀ TRUY CẬP VÀO PHẦN TỬ MẢNG 78](#_Toc145149915)

[PHẦN 10.7 XÓA NỘI DUNG CỦA MẢNG( THIẾT LẬP GIÁ TRỊ BẰNG 0) 79](#_Toc145149916)

[PHẦN 10.8 THIẾT LẬP GIÁ TRỊ TRONG MẢNG 80](#_Toc145149917)

[PHẦN 10.9 CẤP PHÁT VÀ KHỞI TẠO GIÁ TRỊ BAN ĐẦU CHO MỘT MẢNG VỚI KÍCH THƯỚC DO NGƯỜI DÙNG XÁC ĐỊNH 81](#_Toc145149918)

[PHẦN 10.10 DUYỆT QUA MẢNG BẰNG CON TRỎ 82](#_Toc145149919)

[CHƯƠNG 11: LINKED LISTS 83](#_Toc145149920)

[PHẦN 11.1 DOUBLY LINKED LIST (DANH SÁCH LIÊN KẾT KÉP) 83](#_Toc145149921)

[PHẦN 11.2 REVERSING A LINKED LIST (ĐẢO NGƯỢC DANH SÁCH LIÊN KẾT) 86](#_Toc145149922)

[PHẦN 11.3 INSERTING A NODE AT THE NTH POSITION (CHÈN NÚT Ở VỊ TRÍ THỨ N) 89](#_Toc145149923)

[PHẦN 11.4 INSERTING A NODE AT THE BEGINNING OF A SINGLY LINKED LIST (CHÈN NÚT VÀO ĐẦU DANH SÁCH LIÊN KẾT ĐƠN) 90](#_Toc145149924)

[CHƯƠNG 12: ENUMERATIONS 93](#_Toc145149925)

[PHẦN 12.1 SIMPLE ENUMERATION (ENUMERATION ĐƠN GIẢN) 93](#_Toc145149926)

[PHẦN 12.2 ENUMERATION CONSTANT WITHOUT TYPENAME (HẰNG SỐ ENUMERATION KHÔNG CẦN TÊN KIỂU) 95](#_Toc145149927)

[PHẦN 12.3 ENUMERATION WITH DUPLICATE VALUE (ENUMERATION VỚI GIÁ TRỊ TRÙNG LẶP) 95](#_Toc145149928)

[PHẦN 12.4 TYPEDEF ENUM 96](#_Toc145149929)

[CHƯƠNG 13: STRUCTS (CẤU TRÚC) 97](#_Toc145149930)

[PHẦN 13.1 FLEXIBLE ARRAY MEMBERS (THÀNH PHẦN MẢNG LINH HOẠT) 97](#_Toc145149931)

[13.1.1 Khai báo kiểu dữ liệu 97](#_Toc145149932)

[13.1.2 Ảnh hưởng đến kích thước và phần đệm 98](#_Toc145149933)

[13.1.3 Sử dụng 98](#_Toc145149934)

[13.1.4 Struct hack 99](#_Toc145149935)

[13.1.5 Khả năng tương thích 100](#_Toc145149936)

[PHẦN 13.2 TYPEDEF STRUCTS 101](#_Toc145149937)

[PHẦN 13.3 POINTERS TO STRUCTS (CON TRỎ TỚI CẤU TRÚC) 102](#_Toc145149938)

[PHẦN 13.4 PASSING STRUCTS TO FUNCTIONS (TRUYỀN STRUCTS VÀO HÀM) 105](#_Toc145149939)

[PHẦN 13.5 OBJECT- BASED PROGRAMMING USING STRUCTS (LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG SỬ DỤNG STRUCTS) 106](#_Toc145149940)

[PHẦN 13.6 SIMPLE DATA STRUCTURES (CÁC CẤU TRÚC DỮ LIỆU ĐƠN GIẢN) 108](#_Toc145149941)

[CHƯƠNG 14: STANDARD MATH (TOÁN HỌC TIÊU CHUẨN) 109](#_Toc145149942)

[PHẦN 14.1 CÁC HÀM MŨ – POW(), POWF(), POWL() 109](#_Toc145149943)

[PHẦN 14.2 DOUBLE PRECISION FLOATING- POINT REMAINDER FMOD() (PHẦN DƯ SỐ THỰC VỚI ĐỘ CHÍNH XÁC KÉP FMOD()) 111](#_Toc145149944)

[PHẦN 14.3 SINGLE PRECISION AND LONG DOUBLE PRECISION FLOATING- POINT REMAINDER: FMODF(), FMODL() 111](#_Toc145149945)

[CHƯƠNG 15: CÁC LỆNH LẶP/ VÒNG LẶP: FOR, WHILE, DO WHILE 113](#_Toc145149946)

[PHẦN 15.1 VÒNG LẶP FOR 113](#_Toc145149947)

[PHẦN 15.2 LOOP UNROLLING AND DUFF’S DEVICE 114](#_Toc145149948)

[PHẦN 15.3 VÒNG LẶP WHILE 115](#_Toc145149949)

[PHẦN 15.4 VÒNG LẶP DO – WHILE 116](#_Toc145149950)

[PHẦN 15.5 CẤU TRÚC VÀ LUỒNG ĐIỀU KHIỂN TRONG VÒNG LẶP FOR 116](#_Toc145149951)

[PHẦN 15.6 VÒNG LẶP VÔ HẠN 117](#_Toc145149952)

[CHƯƠNG 16: SELECTION STATEMENTS (CÂU LỆNH LỰA CHỌN) 119](#_Toc145149953)

[PHẦN 16.1 CÂU LỆNH IF() 119](#_Toc145149954)

[PHẦN 16.2 NESTED IF()… ELSE VS IF ELSE LADDER ( CÂU LỆNH IF LỒNG NHAU VÀ CÂU LỆNH IF BẬC THANG) 120](#_Toc145149955)

[PHẦN 16.3 CÂU LỆNH SWITCH() 122](#_Toc145149956)

[PHẦN 16.4 CÂU LỆNH IF()… ELSE VÀ CÚ PHÁP 125](#_Toc145149957)

[PHẦN 16.5 NỐI HAI HOẶC NHIỀU LỆNH IF()… ELSE 125](#_Toc145149958)

[CHƯƠNG 17: INITIALIZATION (KHỞI TẠO) 127](#_Toc145149959)

[PHẦN 17.1 KHỞI TẠO BIẾN TRONG C 127](#_Toc145149960)

[17.1.1 Khởi tạo một mảng 128](#_Toc145149961)

[17.1.2 Khởi tạo mảng kí tự 128](#_Toc145149962)

[PHẦN 17.2 SỬ DỤNG DESIGNATED INITIALIZERS 129](#_Toc145149963)

[17.2.1 Designated initializers cho các phần tử mảng 129](#_Toc145149964)

[17.2.2 Designated initializers cho structures 130](#_Toc145149965)

[17.2.3 Designated initializer cho unions 130](#_Toc145149966)

[17.2.4 Designated initializers cho arrays của structures, etc 131](#_Toc145149967)

[17.2.5 Specifying ranges in array initializers (Chỉ định phạm vi trong bộ khởi tạo mảng) 131](#_Toc145149968)

[PHẦN 17.3 INITIALIZING STRUCTURES AND ARRAYS OF STRUCTURES (KHỞI TẠO CẤU TRÚC VÀ MẢNG CẤU TRÚC) 132](#_Toc145149969)

[CHƯƠNG 18: KHAI BÁO VÀ ĐỊNH NGHĨA 133](#_Toc145149970)

[PHẦN 18.1 HIỂU VỀ KHAI BÁO VÀ ĐỊNH NGHĨA 133](#_Toc145149971)

[CHƯƠNG 19: COMMAND- LINE ARGUMENTS (ĐỐI SỐ DÒNG LỆNH) 134](#_Toc145149972)

[PHẦN 19.1 IN RA CÁC ĐỐI SỐ CỦA CHƯƠNG TRÌNH VÀ CHUYỂN ĐỔI THÀNH CÁC GIÁ TRỊ SỐ NGUYÊN 134](#_Toc145149973)

[PHẦN 19.2 IN RA CÁC ĐỐI SỐ CỦA DÒNG LỆNH 135](#_Toc145149974)

[PHẦN 19.3 SỬ DỤNG CÔNG CỤ GETOPT CỦA GNU 136](#_Toc145149975)

[CHƯƠNG 20: FILES VÀ LUỒNG INPUT/OUTPUT 137](#_Toc145149976)

[PHẦN 20.1 MỞ VÀ GHI VÀO TỆP TIN 137](#_Toc145149977)

[PHẦN 20.2 RUN PROCESS (QUÁ TRÌNH CHẠY) 138](#_Toc145149978)

[PHẦN 20.3 FPRINTF 139](#_Toc145149979)

[PHẦN 20.4 LẤY CÁC DÒNG TỪ TỆP SỬ DỤNG GETLINE() 140](#_Toc145149980)

[PHẦN 20.5 FSCANF() 144](#_Toc145149981)

[PHẦN 20.6 ĐỌC CÁC DÒNG TỪ TẬP TIN 145](#_Toc145149982)

[PHẦN 20.7 MỞ VÀ VIẾT VÀO FILE NHỊ PHÂN 147](#_Toc145149983)

[CHƯƠNG 21: FORMATTED INPUT/OUTPUT (ĐỊNH DẠNG ĐẦU VÀO ĐẦU RA) 149](#_Toc145149984)

[PHẦN 21.1 CÁC BỘ CHUYỂN ĐỔI CHO VIỆC IN RA 149](#_Toc145149985)

[PHẦN 21.2 HÀM PRINTF() 151](#_Toc145149986)

[PHẦN 21.3 CÁC CỜ ĐỊNH DẠNG KHI IN 152](#_Toc145149987)

[PHẦN 21.4 IN GIÁ TRỊ CỦA CON TRỎ TỚI ĐỐI TƯỢNG 154](#_Toc145149988)

[PHẦN 21.5 IN GIÁ TRỊ CỦA HIỆU HAI CON TRỎ ĐẾN MỘT ĐỐI TƯỢNG 155](#_Toc145149989)

[PHẦN 21.6 LENGTH MODIFIERS 156](#_Toc145149990)

[CHƯƠNG 22: CON TRỎ 158](#_Toc145149991)

[PHẦN 22.1 GIỚI THIỆU 158](#_Toc145149992)

[PHẦN 22.2 CÁC LỖI THƯỜNG GẶP 161](#_Toc145149993)

[22.2.1 Không kiểm tra lỗi trong việc cấp phát bộ nhớ 161](#_Toc145149994)

[22.2.2 Sử dụng số nguyên cố định thay vì sizeof khi yêu cầu cấp phát bộ nhớ 162](#_Toc145149995)

[22.2.3 Rò rỉ bộ nhớ (Memory leaks) 162](#_Toc145149996)

[22.2.4 Lỗi Logic 162](#_Toc145149997)

[22.2.5 Tạo con trỏ trỏ tới biến được tạo trong vùng nhớ (stack) ngăn xếp 163](#_Toc145149998)

[22.2.6 Tăng giảm và dereferencing 165](#_Toc145149999)

[PHẦN 22.3 DEREFERENCING A POINTER 166](#_Toc145150000)

[PHẦN 22.4 DEREFERENCING A POINTER TO A STRUCT 166](#_Toc145150001)

[PHẦN 22.5 CON TRỎ HẰNG 168](#_Toc145150002)

[22.5.1 Single Pointers 168](#_Toc145150003)

[22.5.2 Pointer to Pointer 169](#_Toc145150004)

[PHẦN 22.6 CON TRỎ HÀM 171](#_Toc145150005)

[PHẦN 22.7 POLYMORPHIC BEHAVIOUR WITH VOID POINTERS 172](#_Toc145150006)

[PHẦN 22.8 TOÁN TỬ ĐỊA CHỈ 173](#_Toc145150007)

[PHẦN 22.9 KHỞI TẠO CON TRỎ 174](#_Toc145150008)

[PHẦN 22.10 POINTER TO POINTER 175](#_Toc145150009)

[PHẦN 22.11 CON TRỎ VOID\* NHƯ LÀ ĐỐI SỐ VÀ GIÁ TRỊ TRẢ VỀ CỦA CÁC HÀM TIÊU CHUẨN 175](#_Toc145150010)

[PHẦN 22.12 CÙNG MỘT DẤU HOA THỊ \* NHƯNG CÓ Ý NGHĨA KHÁC NHAU 176](#_Toc145150011)

[CHƯƠNG 23: SEQUENCE POINTS 178](#_Toc145150012)

[PHẦN 23.1 BIỂU THỨC KHÔNG CÓ THỨ TỰ 178](#_Toc145150013)

[PHẦN 23.2 BIỂU THỨC CÓ THỨ TỰ 179](#_Toc145150014)

[PHẦN 23.3 BIỂU THỨC ĐƯỢC XÁC ĐỊNH THỨ TỰ MỘT CÁCH KHÔNG XÁC ĐỊNH 180](#_Toc145150015)

[CHƯƠNG 24: CON TRỎ HÀM 181](#_Toc145150016)

[PHẦN 24.1 GIỚI THIỆU 181](#_Toc145150017)

[PHẦN 24.2 TRẢ VỀ CON TRỎ HÀM TỪ MỘT HÀM 182](#_Toc145150018)

[PHẦN 24.3 THỰC HÀNH TỐT NHẤT 183](#_Toc145150019)

[PHẦN 24.4 GÁN CON TRỎ HÀM 185](#_Toc145150020)

[PHẦN 24.5 GHI NHỚ ĐỂ VIẾT CON TRỎ HÀM 186](#_Toc145150021)

[PHẦN 24.6 KHÁI NIỆM CƠ BẢN 186](#_Toc145150022)

[CHƯƠNG 25: THAM SỐ CỦA HÀM 188](#_Toc145150023)

[PHẦN 25.1 THAM SỐ ĐƯỢC TRUYỀN BẰNG GIÁ TRỊ (PASS BY VALUE) 188](#_Toc145150024)

[PHẦN 25.2 TRUYỀN MẢNG VÀO HÀM 189](#_Toc145150025)

[PHẦN 25.3 THỨ TỰ THỰC THI THAM SỐ TRONG HÀM 190](#_Toc145150026)

[PHẦN 25.4 SỬ DỤNG THAM SỐ CON TRỎ ĐỂ TRẢ VỀ NHIỀU GIÁ TRỊ 190](#_Toc145150027)

[PHẦN 25.5 VÍ DỤ VỀ HÀM TRẢ VỀ CẤU TRÚC CHỨA CÁC GIÁ TRỊ CÓ MÃ LỖI 191](#_Toc145150028)

[CHƯƠNG 26: TRUYỀN MỘT MẢNG 2D ĐẾN MỘT HÀM 193](#_Toc145150029)

[PHẦN 26.1 TRUYỀN MỘT MẢNG 2D ĐẾN MỘT HÀM 193](#_Toc145150030)

[PHẦN 26.2 SỬ DỤNG MẢNG PHẲNG LÀM MẢNG 2D 201](#_Toc145150031)

[CHƯƠNG 27: XỬ LÝ LỖI 202](#_Toc145150032)

[PHẦN 27.1 ERRNO 202](#_Toc145150033)

[PHẦN 27.2 STRERROR 202](#_Toc145150034)

[PHẦN 27.3 PERROR 203](#_Toc145150035)

[CHƯƠNG 28: UNDEFINED BEHAVIOR (HÀNH VI KHÔNG XÁC ĐỊNH) 204](#_Toc145150036)

[1. DEREFERENCING MỘT CON TRỎ TỚI BIẾN NGOÀI THỜI GIAN TỒN TẠI CỦA NÓ 205](#_Toc145150037)

[2. SAO CHÉP BỘ NHỚ CHỒNG CHÉO 205](#_Toc145150038)

[3. SIGNED INTEGER OVERFLOW (TRÀN SỐ NGUYÊN CÓ DẤU) 206](#_Toc145150039)

[4. SỬ DỤNG MỘT BIẾN CHƯA ĐƯỢC KHỞI TẠO 208](#_Toc145150040)

[5. DATA RACE 209](#_Toc145150041)

[6. ĐỌC GIÁ TRỊ CỦA CON TRỎ ĐÃ ĐƯỢC GIẢI PHÓNG 209](#_Toc145150042)

[7. SỬ DỤNG ĐỊNH DẠNG KHÔNG CHÍNH XÁC TRONG PRINTF 209](#_Toc145150043)

[8. SỬA ĐỔI CHUỖI KÝ TỰ 210](#_Toc145150044)

[9. TRUYỀN MỘT CON TRỎ NULL ĐỂ CHUYỂN ĐỔI PRINTF(%S) 210](#_Toc145150045)

[10. SỬA ĐỔI BẤT KÌ ĐỐI TƯỢNG NÀO NHIỀU LẦN GIỮA HAI SEQUENCE POINTS 211](#_Toc145150046)

[11. GIẢI PHÓNG BỘ NHỚ HAI LẦN 212](#_Toc145150047)

[12. DỊCH CHUYỂN BIT BẰNG CÁCH SỬ DỤNG SỐ ÂM HOẶC VƯỢT QUÁ CHIỀU RỘNG CỦA KIỂU DỮ LIỆU 213](#_Toc145150048)

[13. QUAY LẠI TỪ MỘT HÀM ĐƯỢC KHAI BÁO BẰNG BỘ XÁC ĐỊNH HÀM ‘\_NORETURN’ HOẶC ‘NORETURN’ 213](#_Toc145150049)

[14. TRUY CẬP BỘ NHỚ NGOÀI VÙNG NHỚ ĐƯỢC PHÂN BỔ 214](#_Toc145150050)

[15. THAY ĐỔI BIẾN CONST THÔNG QUA CON TRỎ 215](#_Toc145150051)

[16. ĐỌC MỘT ĐỐI TƯỢNG CHƯA KHỞI TẠO KHÔNG ĐƯỢC HỖ TRỢ BỞI BỘ NHỚ 215](#_Toc145150052)

[17. PHÉP CỘNG HOẶC PHÉP TRỪ CON TRỎ KHÔNG ĐƯỢC GIỚI HẠN ĐÚNG CÁCH 216](#_Toc145150053)

[18. DEREFERENCING CON TRỎ NULL 216](#_Toc145150054)

[19. SỬ DỤNG FFLUSH TRÊN LUỒNG ĐẦU VÀO 217](#_Toc145150055)

[20. LIÊN KẾT KHÔNG NHẤT QUÁN CỦA CÁC ĐỊNH DANH 217](#_Toc145150056)

[21. THIẾU CÂU LỆNH RETURN TRONG CÂU LỆNH TRẢ VỀ GIÁ TRỊ 218](#_Toc145150057)

[22. CHIA CHO SỐ 0 219](#_Toc145150058)

[23. CHUYỂN ĐỔI GIỮA CÁC LOẠI CON TRỎ TẠO RA KẾT QUẢ ĐƯỢC CĂN CHỈNH KHÔNG CHÍNH XÁC 219](#_Toc145150059)

[24. SỬA ĐỔI CHUỖI ĐƯỢC TRẢ VỀ BỞI CÁC HÀM GETENV, STRERROR, VÀ SETLOCALE 220](#_Toc145150060)

[CHƯƠNG 29: TẠO SỐ NGẪU NHIÊN 222](#_Toc145150061)

[1. TẠO SỐ NGẪU NHIÊN CƠ BẢN 222](#_Toc145150062)

[2. TRÌNH TẠO ĐỒNG DƯ ĐƯỢC HOÁN VỊ 223](#_Toc145150063)

[3. THẾ HỆ XORSHIFT 224](#_Toc145150064)

[4. HẠN CHẾ TẠO TRONG MỘT PHẠM VI NHẤT ĐỊNH 224](#_Toc145150065)

[CHƯƠNG 30: TIỀN XỬ LÝ VÀ MACRO 226](#_Toc145150066)

[1. HEADER INCLUDE GUARDS 226](#_Toc145150067)

[2. SỬ DỤNG #IF 0 ĐỂ CHẶN MÃ TRONG CÁC PHẦN 229](#_Toc145150068)

[3. MACROS CÓ HÌNH THỨC CỦA HÀM 230](#_Toc145150069)

[4. CHÈN CÁC TỆP NGUỒN 231](#_Toc145150070)

[5. CONDITIONAL INCLUSION AND CONDITIONAL FUNCTION SIGNATURE MODIFICATION 232](#_Toc145150071)

[6. \_ \_ CPLUSPLUS ĐỂ SỬ DỤNG CÁC PHẦN TỬ C NGOÀI MÃ C ++ KHI BIÊN DỊCH C++ BẰNG TÊN BỊ MÃ HÓA 235](#_Toc145150072)

[7. DÁN MÃ THÔNG BÁO 236](#_Toc145150073)

[8. MACRO ĐƯỢC XÁC ĐỊNH TRƯỚC 236](#_Toc145150074)

[8.1. Macro xác định trước bắt buộc 236](#_Toc145150075)

[8.2. Macro được xác định trước khác (không bắt buộc) 237](#_Toc145150076)

[9. MACRO VỚI ĐỐI SỐ BIẾN THIÊN 238](#_Toc145150077)

[10. THAY THẾ MACRO 240](#_Toc145150078)

[11. CHỈ THỊ LỖI 241](#_Toc145150079)

[12. KHAI TRIỂN FOREACH 241](#_Toc145150080)

[CHƯƠNG 31: XỬ LÝ TÍN HIỆU 245](#_Toc145150081)

[1. XỬ LÝ TÍN HIỆU VỚI “SIGNAL()” 245](#_Toc145150082)

[CHƯƠNG 32: ĐỐI SỐ BIẾN 248](#_Toc145150083)

[1. SỬ DỤNG ĐỐI SỐ ĐẾM RÕ RÀNG ĐỂ XÁC ĐỊNH ĐỘ DÀI CỦA VA\_LIST 248](#_Toc145150084)

[2. SỬ DỤNG GIÁ TRỊ KẾT THÚC ĐỂ XÁC ĐỊNH PHẦN CUỐI CỦA VA\_LIST 249](#_Toc145150085)

[3. TRIỂN KHAI CÁC HÀM VỚI GIAO DIỆN GIỐNG NHƯ ‘ PRINTF() ’ 250](#_Toc145150086)

[4. SỬ DỤNG CHUỖI ĐỊNH DẠNG 253](#_Toc145150087)

[CHƯƠNG 33: KHẲNG ĐỊNH 255](#_Toc145150088)

[1. KHẲNG ĐỊNH ĐƠN GIẢN 255](#_Toc145150089)

[2. KHẲNG ĐỊNH TĨNH 256](#_Toc145150090)

[3. KHẲNG ĐỊNH TIN NHẮN LỖI 257](#_Toc145150091)

[4. KHẲNG ĐỊNH MÃ KHÔNG THỂ TRUY CẬP ĐƯỢC 257](#_Toc145150092)

[5. TIỀN ĐIỀU KIỆN VÀ HẬU ĐIỀU KIỆN 258](#_Toc145150093)

[CHƯƠNG 34: LỰA CHỌN TỔNG QUÁT 260](#_Toc145150094)

[1. KIỂM TRA XEM BIẾN CÓ PHẢI LÀ MỘT KIỂU CỤ THỂ ĐƯỢC CHÚ THÍCH 260](#_Toc145150095)

[2. LỰA CHỌN THÔNG QUA KIỂU DỮ LIỆU CHUNG DỰA TRÊN NHIỀU ĐỐI SỐ 261](#_Toc145150096)

[3. LOẠI MACRO IN CHUNG 262](#_Toc145150097)

[CHƯƠNG 35: X- MACROS 264](#_Toc145150098)

[1. SỬ DỤNG ĐƠN GIẢN CỦA X- MACROS CHO CÁC CÂU LỆNH PRINTF() 264](#_Toc145150099)

[2. MỞ RỘNG : SỬ DỤNG X- MACRO NHƯ LÀ ĐỐI SỐ 264](#_Toc145150100)

[3. GIÁ TRỊ ENUM VÀ MÃ ĐỊNH DANH 265](#_Toc145150101)

[4. TẠO MÃ 265](#_Toc145150102)

[CHƯƠNG 36: ALIASING VÀ EFFECTIVE TYPE 268](#_Toc145150103)

[1. EFFECTIVE TYPE 268](#_Toc145150104)

[2. RESTRICT QUALIFICATION 268](#_Toc145150105)

[3. THAY ĐỔI CÁC BYTE 269](#_Toc145150106)

[4. CÁC KIỂU KÍ TỰ KHÔNG THỂ TRUY CẬP THÔNG QUA CÁC KIỂU KHÔNG PHẢI KIỂU KÍ TỰ 270](#_Toc145150107)

[5. VI PHẠM QUY TẮC “STRICT ALIASING” 271](#_Toc145150108)

[CHƯƠNG 37: BIÊN DỊCH 273](#_Toc145150109)

[1. TRÌNH BIÊN DỊCH (THE COMPILER) 273](#_Toc145150110)

[2. FILE TYPES 274](#_Toc145150111)

[3. THE LINKER (TRÌNH LIÊN KẾT) 275](#_Toc145150112)

[3.1. Liên kết ngầm định của trình liên kết 276](#_Toc145150113)

[3.2. Yêu cầu rõ ràng của trình liên kết 276](#_Toc145150114)

[3.3. Tùy chọn cho trình liên kết 276](#_Toc145150115)

[3.4. Những tùy chọn biên dịch khác 277](#_Toc145150116)

[4. TIỀN XỬ LÝ (THE PREPROCESSOR) 278](#_Toc145150117)

[5. THE TRANSLATION PHASES (CÁC GIAI ĐOẠN DỊCH) 280](#_Toc145150118)

[CHƯƠNG 38: INLINE ASSEMBLY 282](#_Toc145150119)

[1. GCC INLINE ASSEMBLY IN MACROS 282](#_Toc145150120)

[2. HỖ TRỢ ASM CƠ BẢN TRONG GCC 282](#_Toc145150121)

[3. HỖ TRỢ ASM MỞ RỘNG TRONG GCC 283](#_Toc145150122)

[CHƯƠNG 39: PHẠM VI CỦA ĐỊNH DANH 285](#_Toc145150123)

[1. PHẠM VI CỦA KHAI BÁO HÀM 285](#_Toc145150124)

[2. PHẠM VI KHỐI 286](#_Toc145150125)

[3. PHẠM VI CỦA TỆP 287](#_Toc145150126)

[4. PHẠM VI HÀM 287](#_Toc145150127)

[CHƯƠNG 40: CHUYỂN ĐỔI RÕ RÀNG VÀ CHUYỂN ĐỔI NGẦM ĐỊNH 289](#_Toc145150128)

[1. CHUYỂN ĐỔI SỐ NGUYÊN TRONG LỜI GỌI HÀM 289](#_Toc145150129)

[2. CHUYỂN ĐỔI CON TRỎ TRONG LỜI GỌI HÀM 290](#_Toc145150130)

[CHƯƠNG 41: TYPE QUALIFIERS 292](#_Toc145150131)

[1. VOLATILE VARIABLES 292](#_Toc145150132)

[2. BIẾN KHÔNG THỂ SỬA ĐỔI (CONST) 293](#_Toc145150133)

[CHƯƠNG 42: TYPEDEF 295](#_Toc145150134)

[1. TYPEDEF CHO STRUCTURES VÀ UNIONS 295](#_Toc145150135)

[2. TYPEDEF CHO CON TRỎ HÀM 296](#_Toc145150136)

[3. CÁC ỨNG DỤNG ĐƠN GIẢN CỦA TYPEDEF 298](#_Toc145150137)

[3.1. Để đặt tên ngắn gọn cho một kiểu dữ liệu 298](#_Toc145150138)

[3.2. Cải thiện tính di động 298](#_Toc145150139)

[3.3. Để chỉ định cách sử dụng hoặc cải thiện khả năng đọc 299](#_Toc145150140)

[CHƯƠNG 43: LỚP LƯU TRỮ (STORAGE CLASS) 300](#_Toc145150141)

[1. AUTO 300](#_Toc145150142)

[2. REGISTER 300](#_Toc145150143)

[3. STATIC 301](#_Toc145150144)

[4. TYPEDEF 303](#_Toc145150145)

[5. EXTERN 303](#_Toc145150146)

[6. \_THREAD\_LOCAL 304](#_Toc145150147)

[CHƯƠNG 44: KHAI BÁO 306](#_Toc145150148)

[1. GỌI MỘT HÀM TỪ MỘT TỆP C KHÁC 306](#_Toc145150149)

[2. SỬ DỤNG BIẾN TOÀN CỤC 307](#_Toc145150150)

[3. GIỚI THIỆU 308](#_Toc145150151)

[4. TYPEDEF 311](#_Toc145150152)

[5. SỬ DỤNG HẰNG SỐ TOÀN CỤC 312](#_Toc145150153)

[6. SỬ DỤNG QUY TẮC PHẢI TRÁI HOẶC XOẮN ỐC ĐỂ GIẢI MÃ CÁC KHAI BÁO TRONG C 314](#_Toc145150154)

[CHƯƠNG 45: STRUCTURE PADDING VÀ PACKING 319](#_Toc145150155)

[1. PACKING STRUCTURES 319](#_Toc145150156)

[2. STRUCTURE PADDING 320](#_Toc145150157)

[CHƯƠNG 46: MEMORY MANAGEMENT (QUẢN LÝ BỘ NHỚ) 322](#_Toc145150158)

[1. ALLOCATING MEMORY (CẤP PHÁT BỘ NHỚ) 322](#_Toc145150159)

[1.1. Cấp phát tiêu chuẩn 322](#_Toc145150160)

[1.2. Zeroed Memory 323](#_Toc145150161)

[1.3. Aligned Memory 323](#_Toc145150162)

[2. GIẢI PHÓNG BỘ NHỚ (FREEING MEMORY) 324](#_Toc145150163)

[3. THAY ĐỔI KÍCH THƯỚC BỘ NHỚ CẤP PHÁT (REALLOCATING MEMORY) 326](#_Toc145150164)

[4. REALLOC(PTR, 0) KHÔNG TƯƠNG ĐƯƠNG VỚI FREE(PTR) 327](#_Toc145150165)

[5. MẢNG ĐA CHIỀU CÓ KÍCH THƯỚC THAY ĐỔI 327](#_Toc145150166)

[6. ALLOCA: CẤP PHÁT BỘ NHỚ TRÊN STACK 329](#_Toc145150167)

[7. QUẢN LÝ BỘ NHỚ DO NGƯỜI DÙNG ĐỊNH NGHĨA 330](#_Toc145150168)

[CHƯƠNG 47: THỰC HIỆN HÀNH VI ĐƯỢC XÁC ĐỊNH (IMPLEMENTATION – DEFINED BEHAVIOUR) 333](#_Toc145150169)

[1. DỊCH PHẢI MỘT SỐ NGUYÊN ÂM 333](#_Toc145150170)

[2. GÁN MỘT GIÁ TRỊ NẰM NGOÀI PHẠM VI CỦA MỘT SỐ NGUYÊN 333](#_Toc145150171)

[3. CẤP PHÁT SỐ BYTE BẰNG 0 333](#_Toc145150172)

[4. BIỂU DIỄN SỐ NGUYÊN CÓ DẤU 333](#_Toc145150173)

[CHƯƠNG 48: ATOMICS 335](#_Toc145150174)

[1. ATOMICS VÀ CÁC TOÁN TỬ 335](#_Toc145150175)

[CHƯƠNG 49: CÂU LỆNH NHẢY (JUMP STATEMENTS) 336](#_Toc145150176)

[1. SỬ DỤNG RETURN 336](#_Toc145150177)

[2. SỬ DỤNG GOTO ĐỂ NHẢY RA KHỎI CÁC VÒNG LẶP LỒNG NHAU 337](#_Toc145150178)

[3. SỬ DỤNG BREAK VÀ CONTINUE 338](#_Toc145150179)

[CHƯƠNG 50: CREATE AND INCLUDE HEADER FILES 340](#_Toc145150180)

[1. GIỚI THIỆU 340](#_Toc145150181)

[2. SELF – CONTAINMENT 341](#_Toc145150182)

[3. TỐI GIẢN 343](#_Toc145150183)

[4. NOTATION VÀ MISCELLANY 343](#_Toc145150184)

[5. IDEMPOTENCE 345](#_Toc145150185)

[6. INCLUDE WHAT YOU USE (IWYU) 346](#_Toc145150186)

[CHƯƠNG 51: <CTYPE.H> - CHARACTER CLASSFICATION & CONVERTER 348](#_Toc145150187)

[1. GIỚI THIỆU 348](#_Toc145150188)

[2. PHÂN LOẠI CÁC KÍ TỰ ĐỌC TỪ MỘT LUỒNG 350](#_Toc145150189)

[3. PHÂN LOẠI CÁC KÍ TỰ ĐỌC TỪ MỘT CHUỖI 350](#_Toc145150190)

[CHƯƠNG 52: SIDE EFFECTS 352](#_Toc145150191)

[1. TOÁN TỬ TĂNG/GIẢM TRƯỚC/SAU 352](#_Toc145150192)

[CHƯƠNG 53: MULTI – CHARACTER CHARACTER SEQUENCE 354](#_Toc145150193)

[1. TRIGRAPHS 354](#_Toc145150194)

[2. DIGRAPHS 355](#_Toc145150195)

[CHƯƠNG 54: RÀNG BUỘC (CONSTRAINT) 356](#_Toc145150196)

[1. TÊN BIẾN TRÙNG LẶP TRONG CÙNG PHẠM VI 356](#_Toc145150197)

[2. TOÁN TỬ SỐ HỌC MỘT NGÔI 356](#_Toc145150198)

[CHƯƠNG 55: INLINING 357](#_Toc145150199)

[1. HÀM INLINING ĐƯỢC SỬ DỤNG TRONG HƠN MỘT TỆP NGUỒN 357](#_Toc145150200)

[CHƯƠNG 56: UNIONS 360](#_Toc145150201)

[1. SỬ DỤNG UNIONS ĐỂ TÁI TẠO CÁC GIÁ TRỊ 360](#_Toc145150202)

[2. VIẾT VÀO MỘT PHẦN TỬ UNION VÀ ĐỌC TỪ PHẦN TỬ KHÁC 361](#_Toc145150203)

[3. SỰ KHÁC BIỆT GIỮA STRUCT VÀ UNION 361](#_Toc145150204)

[CHƯƠNG 57: THREADS (NATIVE) 363](#_Toc145150205)

[1. KHỞI TẠO MỘT THREAD 363](#_Toc145150206)

[2. BẮT ĐẦU NHIỀU LUỒNG 364](#_Toc145150207)

[CHƯƠNG 58: ĐA LUỒNG (MULTITHREADING) 365](#_Toc145150208)

[1. VÍ DỤ ĐƠN GIẢN VỀ LUỒNG C11 365](#_Toc145150209)

[CHƯƠNG 59: GIAO TIẾP GIỮA CÁC TIẾN TRÌNH (INTERPROCESS COMMUNICATION - IPC) 366](#_Toc145150210)

[1. SEMAPHORES 366](#_Toc145150211)

[CHƯƠNG 60: TESTING FRAMEWORKS 372](#_Toc145150212)

[1. UNITY TEST FRAMEWORK 372](#_Toc145150213)

[2. CMOCKA 373](#_Toc145150214)

[3. CPPUTESTS 374](#_Toc145150215)

[CHƯƠNG 61: VALGRIND 376](#_Toc145150216)

[1. MẤT DỮ LIỆU BYTE – QUÊN GIẢI PHÓNG BỘ NHỚ 376](#_Toc145150217)

[2. CÁC LỖI PHỔ BIẾN THƯỜNG GẶP KHI SỬ DỤNG VALGRIND 376](#_Toc145150218)

[3. RUNNING VALGRIND 378](#_Toc145150219)

[4. ADDING FLAGS 378](#_Toc145150220)

[CHƯƠNG 62: CÁC BIỂU THỨC VÀ THỰC TIỄN PHÁT TRIỂN PHỔ BIẾN TRONG LẬP TRÌNH C 379](#_Toc145150221)

[1. SO SÁNH GIÁ TRỊ CHỮ VÀ BIẾN 379](#_Toc145150222)

[2. ĐỪNG ĐỂ DANH SÁCH THAM SỐ CỦA HÀM TRỐNG – SỬ DỤNG “VOID” 379](#_Toc145150223)

[CHƯƠNG 63: NHỮNG LỖI PHỔ BIẾN 383](#_Toc145150224)

[1. KẾT HỢP SINGED VÀ UNSIGNED INTEGERS TRONG CÁC PHÉP TOÁN SỐ HỌC 383](#_Toc145150225)

[2. MACROS CHỈ LÀ SỰ THAY THẾ CHUỖI ĐƠN GIẢN 384](#_Toc145150226)

[3. QUÊN SAO CHÉP GIÁ TRỊ TRẢ VỀ CỦA REALLOC VÀO MỘT BIẾN TẠM 386](#_Toc145150227)

[4. QUÊN CẤP PHÁT THÊM MỘT BYTE CHO \0 386](#_Toc145150228)

[5. HIỂU SAI VỀ SỰ ARRAY DECAY 387](#_Toc145150229)

[6. QUÊN GIẢI PHÓNG BỘ NHỚ (MEMORY LEAKS) 389](#_Toc145150230)

[7. SAO CHÉP QUÁ NHIỀU 391](#_Toc145150231)

[8. NHẦM LẪN VIẾT = THAY VÌ == KHI SO SÁNH 392](#_Toc145150232)

[9. KÍ TỰ XUỐNG DÒNG KHÔNG ĐƯỢC SƯ DỤNG TRONG LỜI GỌI SCANF() THÔNG THƯỜNG 393](#_Toc145150233)

[10. THÊM DẤU CHẤM PHẨY VÀO #DEFINE 395](#_Toc145150234)

[11. SỬ DỤNG DẤU CHẤM PHẨY KHÔNG CẨN THẬN 395](#_Toc145150235)

[12. LỖI UNDEFINED REFERENCE KHI LINKING 396](#_Toc145150236)

[13. KIỂM TRA BIỂU THỨC LOGIC VỚI TRUE 398](#_Toc145150237)

[14. THỰC HIỆN MỞ RỘNG THÊM TRONG CON TRỎ SỐ HỌC 400](#_Toc145150238)

[15. CHÚ THÍCH NHIỀU DÒNG KHÔNG THỂ LỒNG NHAU 401](#_Toc145150239)

[16. BỎ QUA GIÁ TRỊ TRẢ VỀ CỦA CÁC HÀM THƯ VIỆN 403](#_Toc145150240)

[17. SO SÁNH SỐ DẤU CHẤM ĐỘNG 404](#_Toc145150241)

[18. CHỮ SỐ THẬP PHÂN DẤU CHẤM ĐỘNG MẶC ĐỊNH LÀ KIỂU DOUBLE 406](#_Toc145150242)

[19. SỬ DỤNG HẰNG KÍ TỰ THAY VÌ CHUỖI KÍ TỰ VÀ NGƯỢC LẠI 407](#_Toc145150243)

[20. HÀM ĐỆ QUY – THIẾU ĐIỀU KIỆN CƠ SỞ 408](#_Toc145150244)

[21. VƯỢT QUA GIỚI HẠN MẢNG 409](#_Toc145150245)

[22. TRUYỀN MẢNG KHÔNG KỀ NHAU VÀO HÀM MONG ĐỢI MẢNG ĐA CHIỀU “THỰC SỰ” 410](#_Toc145150246)

# BẮT ĐẦU VỚI NGÔN NGỮ C

|  |  |  |
| --- | --- | --- |
| **Phiên bản** | **Tiêu chuẩn** | **Ngày xuất bản** |
| K&R | n/a | 1978-02-22 |
| C89 | ANSI X3.159-1989 | 1989-12-14 |
| C90 | ISO/IEC 9899:1990 | 1990-12-20 |
| C95 | ISO/IEC 9899/AMD1:1995 | 1995-03-30 |
| C99 | ISO/IEC 9899:1999 | 1999-12-16 |
| C11 | ISO/IEC 9899:2011 | 2011-12-15 |

## Hello World

Để tạo một chương trình C đơn giản in dòng chữ "Hello World " lên màn hình, sử dụng trình soạn thảo văn bản để tạo file mới (ví dụ: hello.c — phần mở rộng file phải là .c) chứa mã nguồn sau:

**hello.c**

|  |
| --- |
| # include<stdio.h>  int main(void)  {  puts("Hello World");  return 0;  } |

**Hãy xem từng dòng trong chương trình đơn giản này**

|  |
| --- |
| # include<stdio.h> |

* Dòng mã này yêu cầu trình biên dịch cần thêm nội dung của tệp tiêu đề thư viện chuẩn **stdio.h** vào chương trình. Tệp tiêu đề thường chứa các khai báo của các hàm, macros và kiểu dữ liệu và bạn cần phải thêm vào tệp tiêu đề trước khi sử dụng chúng. Dòng này thêm vào thư viện **stdio.h** để nó có thể gọi hàm **puts().**

|  |
| --- |
| int main(void) |

* Dòng mã này bắt đầu định nghĩa của một hàm. Nó cho biết tên của hàm (main), kiểu dữ liệu và số lượng các đối số mà nó mong đợi (void, nghĩa là không có) và loại giá trị mà hàm này trả về (int). Theo nguyên tắc, **main** là một tên đặc biệt, nó chỉ dùng để đặt cho function chính của chương trình, và lúc nào việc thực thi chương trình cũng sẽ bắt đầu từ hàm **main**.

|  |
| --- |
| {  ...  } |

* Dấu ngoặc nhọn (curly braces) được sử dụng thành cặp để chỉ định vị trí bắt đầu và kết thúc một khối mã (khối lệnh). Chúng có thể được sử dụng theo nhiều cách khác nhau, nhưng trong trường hợp này, chúng chỉ định vị trí bắt đầu và kết thúc của một hàm.

|  |
| --- |
| puts("Hello, World"); |

* Dòng này gọi hàm puts() để đưa ra văn bản ra đầu ra tiêu chuẩn (mặc định là màn hình), tiếp theo là một dòng mới.
* Chuỗi cần đưa ra được bao gồm trong cặp dấu ngoặc đơn(). "Hello, World" là chuỗi sẽ được ghi ra màn hình. Trong ngôn ngữ C, mọi giá trị chuỗi phải nằm trong dấu ngoặc kép "..."
* Trong các chương trình C, mỗi câu lệnh cần được kết thúc bằng dấu chấm phẩy (tức là **;**).

|  |
| --- |
| return 0; |

* Khi chúng ta định nghĩa hàm **main(),** chúng ta khai báo nó là một hàm trả về kiểu **int**, có nghĩa là nó cần trả về một giá trị số nguyên. Trong ví dụ này, chúng ta đang trả về giá trị nguyên là **0**, được sử dụng để chỉ rằng chương trình đã kết thúc thành công. Sau câu lệnh **return 0;**, quá trình thực thi sẽ kết thúc.

## Chỉnh sửa chương trình

* Các trình soạn thảo, chỉnh sửa văn bản đơn giản bao gồm **vim** hoặc **gedit** trên **Linux,** hoặc **Notepad** trên **Windows.** Các trình soạn thảo đa nền tảng khác bao gồm **Visual Studio Code** hoặc **Sublime Text.**
* Trình chỉnh sửa phải tạo các tệp văn bản thuần túy, không phải RTF hoặc bất kỳ định dạng nào khác.

## Biên dịch và chạy chương trình

Để chạy chương trình, file nguồn này (hello.c) trước tiên cần được biên dịch thành file thực thi (ví dụ: **hello** trên hệ thống **Unix/Linux** hoặc **hello.exe** trên **Windows**). Điều này được thực hiện bằng cách sử dụng trình biên dịch cho ngôn ngữ C.

### Biên dịch bằng GCC

GCC **(GNU Compiler Collection)** là một trình biên dịch C được sử dụng rộng rãi. Để sử dụng nó, hãy mở **Terminal**, sử dụng dòng lệnh để điều hướng đến vị trí của tệp nguồn rồi chạy câu lệnh sau:

|  |
| --- |
| gcc hello.c -o hello |

Nếu không tìm thấy lỗi trong mã nguồn (hello.c), trình biên dịch sẽ tạo một tệp nhị phân, tên của tệp được cung cấp bởi đối số cho tùy chọn dòng lệnh -o (hello). Đây là tập tin thực thi cuối cùng.

Chúng tôi cũng có thể sử dụng các tùy chọn cảnh báo *-Wall -Wextra -Werror*, giúp xác định các sự cố có thể khiến chương trình bị lỗi hoặc tạo ra kết quả không mong muốn. Chúng không cần thiết cho chương trình đơn giản này nhưng đây là cách thêm chúng:

|  |
| --- |
| gcc -Wall -Wextra -Werror -o hello hello.c |

### Sử dụng trình biên dịch clang

Để biên dịch chương trình bằng **clang**, bạn có thể sử dụng:

|  |
| --- |
| clang -Wall -Wextra -Werror -o hello hello.c |

Theo thiết kế, các tùy chọn dòng lệnh **clang** tương tự như của **GCC.**

### Sử dụng trình biên dịch Microsoft C từ dòng lệnh

Nếu sử dụng trình biên dịch Microsoft cl.exe trên hệ thống Windows hỗ trợ Visual Studio và nếu tất cả các biến môi trường được đặt, ví dụ C này có thể được biên dịch bằng lệnh sau. Lệnh này sẽ tạo ra tệp hello.exe có thể thực thi được trong thư mục mà lệnh được thực thi trong (Có các tùy chọn cảnh báo như /W3 cho cl, đại khái là tương tự như -Wall vv cho GCC hoặc clang).

|  |
| --- |
| cl hello.c |

### Thực hiện chương trình

Sau khi được biên dịch, tệp nhị phân sau đó có thể được thực thi bằng cách nhập ./hello trong **Terminal**. Khi thực thi, chương trình đã biên dịch sẽ in **Hello World**, theo sau là một dòng mới, tới dấu nhắc lệnh.

## “HELLO WORLD!” trong K&R C

Sau đây là bản gốc "Hello World!" chương trình từ cuốn sách **The C Programming Language** của BrianKernighan và Dennis Ritchie (Ritchie là nhà phát triển ban đầu của ngôn ngữ lập trình C tại Bell Labs), được gọi là "K&R":

**Version = K&R**

|  |
| --- |
| #include <stdio.h>  main() {  printf("hello world\n");  } |

Lưu ý rằng **The C Programming Language** không được chuẩn hóa vào thời điểm viết ấn bản đầu tiên của cuốn sách này (1978), và chương trình này có thể sẽ không được biên dịch trên hầu hết các trình biên dịch hiện đại trừ khi chúng được hướng dẫn chấp nhận mã C90.

Câu ví dụ đầu tiên trong cuốn sách K&R (**The C Programming Language**) hiện được coi là chất lượng kém, một phần là do thiếu một kiểu trả về rõ ràng cho **main()** và một phần là do thiếu câu lệnh **return**. Phiên bản thứ hai của cuốn sách được viết cho tiêu chuẩn C89 cũ. Trong C89, kiểu của main sẽ mặc định là int, nhưng ví dụ K&R không trả về một giá trị xác định cho môi trường. Trong các tiêu chuẩn C99 và sau này, kiểu trả về là bắt buộc, nhưng có thể bỏ qua câu lệnh **return** của main (**và chỉ main**), vì có một trường hợp đặc biệt được giới thiệu với ***C99 5.1.2.2.3*** - nó tương đương với việc trả về 0, biểu thị thành công.

Dạng chính được khuyến nghị và di động nhất cho các hệ thống được lưu trữ là **int main(void)** khi chương trình không sử dụng bất kỳ đối số dòng lệnh nào, hoặc **int main(int argc, char \*\*argv)** khi chương trình sử dụng các đối số ở các dòng lệnh.

### C90 §5.1.2.2.3 Kết thúc chương trình

* Một câu lệnh return từ lời gọi ban đầu tới hàm main tương đương với việc gọi hàm exit với giá trị trả về của hàm main là đối số. Nếu hàm main thực hiện một câu lệnh return mà không xác định giá trị, trạng thái kết thúc trả về cho môi trường chủ không được xác định.

### C90 §6.6.6.4 Câu lệnh return

* Nếu một câu lệnh return mà không có biểu thức được thực thi và giá trị của lời gọi hàm được sử dụng bởi người gọi, hành vi đó là không xác định. Đạt đến dấu } kết thúc một hàm tương đương với việc thực thi một câu lệnh return mà không có biểu thức.

### C99 §5.1.2.2.3 Kết thúc chương trình

* Nếu kiểu trả về của hàm main là một kiểu tương thích với int, một câu lệnh return từ lời gọi ban đầu tới hàm main tương đương với việc gọi hàm exit với giá trị trả về của hàm main là đối số; đạt đến dấu } kết thúc hàm main sẽ trả về giá trị 0. Nếu kiểu trả về không tương thích với int, trạng thái kết thúc trả về cho môi trường chủ không được xác định.

# CHÚ THÍCH

Chú thích (comments) được sử dụng để chỉ điều gì đó cho người đọc mã nguồn. Chú thích được coi như một phần trống trên mã nguồn và không thay đổi bất kỳ ý nghĩa thực sự nào của mã. Có hai cú pháp được sử dụng cho chú thích trong ngôn ngữ C, đó là **/\* \*/** (bản gốc) và **//** (bản mới hơn 1 chút). Một số hệ thống tài liệu sử dụng chú thích được định dạng đặc biệt để hỗ trợ tạo tài liệu cho mã nguồn.

## Chú thích bằng cách sử dụng tiền xử lý (Preprocessor)

Một phần lớn mã nguồn cũng có thể bị "chú thích" bằng cách sử dụng các chỉ thị tiền xử lý **#if 0** và **#endif.** Điều này hữu ích khi mã chứa các chú thích nhiều dòng mà không thể lồng nhau nếu không có cơ chế chú thích này.

|  |
| --- |
| #if 0 /\* Bắt đầu "chú thích", bất cứ điều gì từ đây trở đi sẽ bị loại bỏ bởi tiền xử lý \*/  /\* Một lượng mã lớn với các chú thích nhiều dòng \*/  int foo() {  /\* một số câu lệnh \*/  ...  /\* ... một chú thích mô tả câu lệnh if ... \*/  if (someTest) {  /\* some more comments \*/  return 1;  }  return 0;  }  #endif /\* 0 \*/  /\* Mã từ đây trở đi không bị "chú thích" (được bao gồm trong tệp thực thi biên dịch) \*/ |

## Chú thích được giới hạn bằng /\* \*/

Một chú thích bắt đầu với một dấu gạch chéo kề ngay sau đó là một dấu hoa thị (/\*) và kết thúc ngay khi gặp một dấu hoa thị kề ngay sau đó là một dấu gạch chéo (\*/). Mọi thứ nằm giữa các cặp ký tự này đều là chú thích và được coi là một phần trống (tức là bị bỏ qua) bởi trình biên dịch.

|  |
| --- |
| /\* Đây là một chú thích \*/ |

Chú thích ở trên là chú thích trên một dòng. Chú thích kiểu /\* này có thể trải dài trên nhiều dòng, như sau:

|  |
| --- |
| /\* Đây là một chú thích  nhiều dòng \*/ |

Mặc dù điều này không bắt buộc, quy ước phong cách thông thường với chú thích trên nhiều dòng là đặt khoảng trắng và dấu hoa thị ở các dòng sau đầu tiên và đặt /\* và \*/ trên các dòng mới, sao cho chúng xếp thành hàng:

|  |
| --- |
| /\* Đây là  \* một chú  \* thích  \* nhiều dòng  \*/ |

Những dấu hoa thị thêm không có tác dụng chức năng nào đối với chú thích vì chúng không có dấu gạch chéo liên quan.

Các chú thích kiểu /\* này có thể được sử dụng trên một dòng riêng biệt, ở cuối một dòng mã hoặc ngay cả trong các dòng mã:

|  |
| --- |
| /\* Chú thích này nằm trên một dòng riêng biệt \*/  if(x && y) { /\* Chú thích này nằm ở cuối dòng mã \*/  if ((complexCondition1) /\* Chú thích này nằm trong một dòng mã \*/  && (complexCondition2)) {  /\* Chú thích này nằm trong một câu lệnh if, trên một dòng riêng biệt \*/  }  } |

Chú thích không thể lồng nhau. Điều này xảy ra vì bất kỳ /\* nào tiếp theo sẽ bị bỏ qua (là một phần của chú thích) và đầu tiên \*/ gặp được sẽ được coi là kết thúc chú thích. Chú thích trong ví dụ dưới đây sẽ không hoạt động:

|  |
| --- |
| /\* Chú thích bên ngoài, có nghĩa là điều này bị bỏ qua => /\* chú thích nội bộ \*/ <= kết thúc chú thích, không phải chú thích này => \*/ |

Để chú thích các khối mã chứa chú thích kiểu này, mà nếu không sẽ bị lồng nhau, bạn có thể tham khảo ví dụ về Chú thích bằng cách sử dụng tiền xử lý.

## Chú thích được giới hạn bằng dấu //

***Phiên bản ≥ C99***

C99 đã giới thiệu việc sử dụng chú thích trên một dòng theo kiểu C++. Loại chú thích này bắt đầu bằng hai dấu gạch chéo kề nhau và kéo dài đến cuối dòng:

|  |
| --- |
| // Mỗi dòng trong số này đều là một chú thích trên một dòng  // Chú ý rằng mỗi dòng phải bắt đầu bằng  // hai dấu gạch chéo kép liên tiếp (//) |

Loại chú thích này có thể được sử dụng trên một dòng riêng biệt hoặc ở cuối một dòng mã. Tuy nhiên, do chú thích này kéo dài đến cuối dòng, nên chúng không thể được sử dụng trong một dòng mã.

|  |
| --- |
| // Chú thích này nằm trên một dòng riêng biệt  if (x && y) { // Chú thích này nằm ở cuối dòng  // Chú thích này nằm trong một câu lệnh if, trên một dòng riêng biệt  } |

## Rủi ro có thể xảy ra do TRIGRAPH

***Version ≥ C99***

Trong quá trình viết chú thích được giới hạn bởi //, có thể xảy ra lỗi chính tả ảnh hưởng đến hoạt động dự kiến của chú thích. Nếu bạn gõ sai như sau:

|  |
| --- |
| int foo = 20; // Bắt đầu từ 20 ??/ |

Dấu gạch chéo / ở cuối là một lỗi chính tả nhưng giờ đây sẽ được hiểu là dấu gạch chéo ngược . Điều này xảy ra vì **??/** tạo thành một trigraph. **Trigraph ??/** thực tế là một cách viết dài cho dấu gạch chéo ngược , đây là ký hiệu tiếp tục dòng. Điều này có nghĩa là trình biên dịch sẽ hiểu rằng dòng tiếp theo là tiếp tục của dòng hiện tại, tức là tiếp tục của chú thích, điều này có thể không phải ý định ban đầu.

|  |
| --- |
| int foo = 20; // Bắt đầu từ 20 ??/  int bar = 0;  // Dòng sau sẽ gây lỗi biên dịch (biến 'bar' chưa được khai báo)  // vì 'int bar = 0;' là một phần của chú thích trên dòng trước đó  bar += foo; |

# KIỂU DỮ LIỆU

## Giải thích các khai báo

Một đặc điểm cú pháp đặc biệt của ngôn ngữ C là các khai báo phản ánh việc sử dụng của đối tượng được khai báo như trong một biểu thức thông thường.

Tập hợp các toán tử sau đây có cùng mức độ ưu tiên và tính kết hợp được tái sử dụng trong các khai báo, bao gồm:

* Toán tử unary \* "dereference" (giải tham chiếu) thể hiện một con trỏ.
* Toán tử binary [] "array subscription" (phần tử mảng) thể hiện một mảng.
* Toán tử (1+n)-ary () "function call" (gọi hàm) thể hiện một hàm.
* Dấu ngoặc () nhóm nhưng ghi đè mức độ ưu tiên và tính kết hợp của các toán tử còn lại được liệt kê.

Ba toán tử trên có độ ưu tiên và tính kết hợp như sau:

|  |  |  |
| --- | --- | --- |
| Toán tử | Mức độ ưu tiên | Tính kết hợp |
| [ ] Phần tử mảng | 1 | Trái sang phải |
| ( ) Gọi hàm | 1 | Trái sang phải |
| \*Giải tham chiếu | 2 | Phải sang trái |

Khi giải thích các khai báo, người đọc cần bắt đầu từ từng định danh và áp dụng các toán tử kề cạnh theo thứ tự đúng như trong bảng trên. Mỗi lần áp dụng một toán tử, ta có thể thay thế nó bằng các từ tiếng Anh sau đây:

|  |  |
| --- | --- |
| Biểu thức | Giải thích |
| Thing[X] | Kích thức của mảng X là… |
| Thing(x1,x2,x3) | 1 hàm nhận x1 x2 x3 và trả về … |
| \*Thing | Một con trỏ trỏ tới .. |

Theo đó phần bắt đầu của việc giải thích bằng tiếng Anh sẽ luôn bắt đầu với định danh và kết thúc với kiểu dữ liệu nằm ở phía bên trái của khai báo.

**Ví dụ:**

|  |
| --- |
| char \*names[20]; |

[] được ưu tiên hơn \*, vì vậy cách hiểu là: **names** là một mảng có kích thước 20 của một con trỏ trỏ tới char.

|  |
| --- |
| char (\*place)[10]; |

Trong trường hợp sử dụng dấu ngoặc () để ghi đè mức độ ưu tiên, toán tử \* được áp dụng trước: **place** là một con trỏ tới một mảng có kích thước là 10 của char.

|  |
| --- |
| int fn(long, short); |

Ở đây không có ưu tiên cần quan tâm: fn là một hàm nhận tham số long và short, và trả về kiểu int.

|  |
| --- |
| int \*fn(void); |

Dấu ngoặc () được áp dụng trước: fn là một hàm nhận tham số void và trả về một con trỏ tới kiểu int.

|  |
| --- |
| int (\*fp)(void); |

Ghi đè mức độ ưu tiên của (): fp là một con trỏ tới một hàm nhận tham số void và trả về kiểu int.

|  |
| --- |
| int arr[5][8]; |

Mảng đa chiều không phải là một ngoại lệ trong quy tắc; các toán tử [] được áp dụng theo thứ tự từ trái sang phải theo tính kết hợp được liệt kê trong bảng: arr là một mảng có kích thước 5 của một mảng có kích thước 8 của kiểu int.

|  |
| --- |
| int \*\*ptr; |

Hai toán tử dereference có cùng mức độ ưu tiên, do đó tính kết hợp có hiệu lực. Các toán tử được áp dụng theo thứ tự từ phải sang trái: ptr là một con trỏ tới một con trỏ tới kiểu int.

**Khai báo nhiều biến**

Dấu phẩy có thể được sử dụng làm dấu phân tách (không hoạt động như toán tử dấu phẩy) để phân cách các khai báo đa biến trong cùng một câu lệnh. Câu lệnh sau chứa năm khai báo:

|  |
| --- |
| int fn(void), \*ptr, (\*fp)(int), arr[10][20], num; |

Các đối tượng được khai báo trong ví dụ trên là:

* fn: một hàm không tham số và trả về kiểu int.
* ptr: một con trỏ tới kiểu int.
* fp: một con trỏ hàm nhận tham số kiểu int và trả về kiểu int.
* arr: một mảng có kích thước 10, mỗi phần tử là một mảng có kích thước 20 kiểu int.
* num: kiểu int.

**Giải thích thay thế**

Bởi vì các khai báo phản ánh việc sử dụng, nên một khai báo cũng có thể được giải thích dựa trên các toán tử có thể được áp dụng lên đối tượng và kiểu dữ liệu cuối cùng của biểu thức đó. Kiểu dữ liệu nằm ở phía bên trái là kết quả cuối cùng sau khi áp dụng tất cả các toán tử.

|  |
| --- |
| /\*  \*Sử dụng toán tử [] để lấy phần tử của "arr" và sau đó dùng toán tử dereference để lấy giá trị,  \*ta được một kết quả kiểu "char".  \*Cụ thể: \*arr[5] có kiểu "char".  \*/  char arr[20];  /\*  \*Gọi hàm "fn" trả về một kết quả kiểu "int".  \*Cụ thể: fn('b') có kiểu "int".  \*/  int fn(char);  /\*  \*Sử dụng toán tử dereference để lấy giá trị của "fp", sau đó gọi nó trả về một kết quả kiểu "int".  \*Cụ thể: (\*fp)() có kiểu "int".  \*/  int (fp)(void);  /\*  \*Sử dụng toán tử [] hai lần để lấy phần tử của "strings", sau đó sử dụng toán tử dereference để lấy giá trị,  \*ta được một kết quả kiểu "char".  \*Cụ thể: \*strings[5][15] có kiểu "char".  \*/  char \*strings[10][20]; |

## Kiểu số nguyên có độ rộng cố định ( kể từ c99)

Phiên bản ≥ C99 Tiêu đề <stdint.h> cung cấp một số định nghĩa kiểu số nguyên có độ rộng cố định. Các kiểu này là tùy chọn và chỉ được cung cấp nếu nền tảng có một kiểu số nguyên có độ rộng tương ứng và nếu kiểu có dấu tương ứng có biểu diễn bù hai của các giá trị âm. Xem phần lưu ý để biết các gợi ý về việc sử dụng các kiểu có độ rộng cố định.

|  |
| --- |
| /\* Các kiểu phổ biến thường được sử dụng bao gồm \*/  *uint32\_t* u32 = 32; / có độ rộng chính xác 32 bit /  uint8\_t u8 = 255; / có độ rộng chính xác 8 bit /  int64\_t i64 = -65; / có độ rộng chính xác 64 bit trong biểu diễn bù hai \*/ |

## Kiểu số nguyên và hằng số

Các số nguyên có dấu có thể thuộc các kiểu sau đây (int sau từ short hoặc long là tùy chọn):

|  |
| --- |
| signed char c = 127; /\* yêu cầu có độ rộng 1 byte,  xem phần lưu ý để biết thêm thông tin \*/  signed short int si = 32767; /\* yêu cầu có ít nhất 16 bit \*/  signed int i = 32767; /\* yêu cầu có ít nhất 16 bit \*/  signed long int li = 2147483647; /\* yêu cầu có ít nhất 32 bit \*/  Phiên bản ≥ C99  signed long long int li = 2147483647; /\* yêu cầu có ít nhất 64 bit \*/ |

Mỗi kiểu số nguyên có dấu này đều có một phiên bản không dấu tương ứng.

|  |
| --- |
| unsigned int i = 65535;  unsigned short = 2767;  unsigned char = 255; |

Đối với tất cả các kiểu, ngoại trừ char, phiên bản có dấu được cho là mặc định nếu phần có dấu hoặc không dấu bị bỏ qua. Kiểu char tạo thành một kiểu ký tự thứ ba, khác với signed char và unsigned char, và sự có dấu (hoặc không) phụ thuộc vào nền tảng.

Có các loại hằng số số nguyên khác nhau (được gọi là chữ số trong ngôn ngữ C) có thể được viết theo các cơ số khác nhau và có độ rộng khác nhau, dựa trên tiền tố hoặc hậu tố của chúng.

|  |
| --- |
| /\* Các biến sau được khởi tạo với cùng một giá trị: \*/  int d = 42; /\* hằng số thập phân (cơ số 10) \*/  int o = 052; /\* hằng số bát phân (cơ số 8) \*/  int x = 0xaf; /\* hằng số thập lục phân (cơ số 16) \*/  int X = 0XAf; /\*\*/ (chữ cái 'a' đến 'f' (không phân biệt chữ hoa chữ thường) biểu thị từ 10 đến 15) \*/ |

Hằng số thập phân luôn luôn có dấu**(signed)**. Hằng số thập lục phân bắt đầu bằng 0x hoặc 0X và hằng số bát phân bắt đầu chỉ với chữ số 0. Hai loại hằng số sau có thể có dấu**(signed)** hoặc không dấu**(unsigned)** tùy thuộc vào việc giá trị có phù hợp với kiểu có dấu hay không.

|  |
| --- |
| /\* Tiền tố để chỉ độ rộng và dấu \*/  long int i = 0x32; /\* không có tiền tố đại diện cho int hoặc long int \*/  unsigned int ui = 65535u; /\* u hoặc U đại diện cho unsigned int hoặc unsigned long int \*/  long int li = 65536l; /\* l hoặc L đại diện cho long int \*/ |

Nếu không có tiền tố, hằng số có kiểu dữ liệu là kiểu đầu tiên mà giá trị của nó phù hợp, nghĩa là hằng số thập phân lớn hơn INT\_MAX sẽ có kiểu long nếu có thể, hoặc long long nếu không được.

Tập tin tiêu đề <limits.h> mô tả các giới hạn của số nguyên như sau. Các giá trị xác định bởi người thực hiện (implementation-defined) phải có giá trị tuyệt đối lớn hơn hoặc bằng giá trị được hiển thị dưới đây, với cùng dấu.

|  |  |  |
| --- | --- | --- |
| MARCO | Kiểu dữ liệu | Giá trị |
| CHAR\_BIT | smallest object that is not a bit-field (byte) 8 |  |
| SCHAR\_MIN | signed char | -127 / -(27 - 1) |
| SCHAR\_MAX | signed char | +127 / 27 - 1 |
| UCHAR\_MAX | unsigned char | 255 / 28 - 1 |
| CHAR\_MIN | char | see below |
| CHAR\_MAX | char | see below |
| SHRT\_MIN | short int | -32767 / -(215 - 1) |
| SHRT\_MAX | short int | +32767 / 215 - 1 |
| USHRT\_MAX | unsigned short int | 65535 / 216 - 1 |
| INT\_MIN | int | -32767 / -(215 - 1) |
| INT\_MAX | int | +32767 / 215 - 1 |
| UINT\_MAX | unsigned int | 65535 / 216 - 1 |
| LONG\_MIN | long int | -2147483647 / -(231 - 1) |
| LONG\_MAX | long int | +2147483647 / 231 - 1 |
| ULONG\_MAX | unsigned long int | 4294967295 / 232 - 1 |
| Version ≥ C99 |  |  |
| LLONG\_MIN | long long int | -9223372036854775807 / -(263 - 1) |
| LLONG\_MAX | long long int | +9223372036854775807 / 263 - 1 |
| ULLONG\_MAX | unsigned long long int | 18446744073709551615 / 264 - 1 |

Nếu giá trị của một đối tượng có kiểu char được mở rộng dấu khi sử dụng trong một biểu thức, giá trị của CHAR\_MIN sẽ giống như SCHAR\_MIN và giá trị của CHAR\_MAX sẽ giống như SCHAR\_MAX. Nếu giá trị của một đối tượng có kiểu char không được mở rộng dấu khi sử dụng trong một biểu thức, giá trị của CHAR\_MIN sẽ là 0 và giá trị của CHAR\_MAX sẽ giống như UCHAR\_MAX.

Phiên bản ≥ C99

Tiêu chuẩn C99 đã thêm một tiêu đề mới, <stdint.h>, chứa các định nghĩa cho các số nguyên có chiều rộng cố định. Xem ví dụ số nguyên chiều rộng cố định để được giải thích sâu hơn.

## Hằng số thực

Ngôn ngữ C có ba kiểu số thực dấu phẩy động bắt buộc, đó là **float**, **double** và **long double**.

|  |
| --- |
| float f = 0.314f; /\* Hậu tố f hoặc F chỉ định kiểu float \*/  double d = 0.314; /\* Không có hậu tố chỉ định kiểu double \*/  long double ld = 0.314l; /\* Hậu tố l hoặc L chỉ định kiểu long double \*/  /\* Các phần khác nhau trong một định nghĩa số thực là tùy chọn \*/  double x = 1.; /\* Hợp lệ, phần thập phân là tùy chọn \*/  double y = .1; /\* Hợp lệ, phần nguyên là tùy chọn \*/  /\* Chúng cũng có thể được định nghĩa trong dạng ký hiệu khoa học \*/  double sd = 1.2e3; /\* Phần thập phân 1.2 được nhân với 10^3, tức là 1200.0 \*/ |

Tiêu đề <float.h> xác định các giới hạn khác nhau cho các hoạt động của dấu phẩy động Số học dấu phẩy động được xác định triển khai. Tuy nhiên, hầu hết các nền tảng hiện đại (arm, x86, x86\_64, MIPS) đều sử dụng các hoạt động của dấu phẩy động IEEE 754. C cũng có ba loại dấu phẩy động phức tạp tùy chọn bắt nguồn từ loại trên.

## Chuỗi kí tự

Một chuỗi ký tự trong ngôn ngữ C là một chuỗi các ký tự, kết thúc bằng một ký tự số không (literal zero)

|  |
| --- |
| char\* str = "hello, world"; /\* chuỗi ký tự nguyên mẫu (string literal) \*/  /\* chuỗi ký tự nguyên mẫu có thể được sử dụng để khởi tạo mảng \*/  char a1[] = "abc"; /\* a1 là mảng char[4] chứa {'a','b','c','\0'} \*/  char a2[4] = "abc"; /\* tương tự a1 \*/  char a3[3] = "abc"; /\* a3 là mảng char[3] chứa {'a','b','c'}, thiếu ký tự '\0' \*/ |

Chuỗi ký tự nguyên mẫu trong C **không thể sửa đổi**, và việc cố gắng thay đổi giá trị của chúng sẽ dẫn đến hành vi không xác định. Chuỗi ký tự nguyên mẫu thường được lưu trữ trong bộ nhớ chỉ đọc (read-only memory), chẳng hạn như phần .rodata, và mọi cố gắng sửa đổi giá trị của chúng có thể dẫn đến kết quả không mong muốn.

|  |
| --- |
| char\* s = "foobar";  s[0] = 'F'; /\* hành vi không xác định \*/  /\* Thực hành tốt là đánh dấu chuỗi nguyên mẫu là hằng số bằng cách sử dụng const \*/  char const s1 = "foobar";  s1[0] = 'F'; /\* lỗi biên dịch! \*/ |

Nhiều chuỗi ký tự được nối vào thời gian biên dịch, có nghĩa là bạn có thể viết cấu trúc như thế này.

|  |
| --- |
| Phiên bản trước C99  /\* Chỉ có thể nối hai chuỗi ký tự hẹp (narrow) hoặc hai chuỗi ký tự rộng (wide) \*/  char\* s = "Hello, " "World";  Phiên bản C99 trở lên  /\* Từ phiên bản C99 trở đi, có thể nối nhiều hơn hai chuỗi \*/  /\* Quá trình nối chuỗi phụ thuộc vào triển khai cụ thể \*/  char\* s1 = "Hello" ", " "World";  /\* Một số cách sử dụng phổ biến là nối chuỗi định dạng (format strings) \*/  char\* fmt = "%" PRId16; /\* Macro PRId16 được định nghĩa từ C99 \*/ |

Chuỗi ký tự nguyên mẫu, tương tự như hằng số ký tự, hỗ trợ các bộ ký tự khác nhau.

|  |
| --- |
| /\* Chuỗi ký tự thông thường, kiểu char[] \*/  char\* s1 = "abc";  /\* Chuỗi ký tự rộng (wide character), kiểu wchar\_t[] \*/  *wchar\_t*\* s2 = L"abc";  Phiên bản C11 trở lên  /\* Chuỗi ký tự UTF-8, kiểu char[] \*/  char\* s3 = u8"abc";  /\* Chuỗi ký tự rộng 16-bit, kiểu char16\_t[] \*/  *char16\_t*\* s4 = u"abc";  /\* Chuỗi ký tự rộng 32-bit, kiểu char32\_t[] \*/  *char32\_t*\* s5 = U"abc"; |

# TOÁN TỬ

Toán tử trong ngôn ngữ lập trình là một ký hiệu cho biết trình biên dịch hoặc trình thông dịch thực hiện một phép toán, phép toán quan hệ hoặc logic cụ thể và tạo ra kết quả cuối cùng

Ngôn ngữ C có nhiều toán tử mạnh mẽ. Nhiều toán tử trong C là toán tử nhị phân, có nghĩa là chúng có hai toán hạng. Ví dụ, trong biểu thức a / b, / là một toán tử nhị phân nhận hai toán hạng (a, b). Ngoài ra, còn có một số toán tử một ngôi (unary operators) chỉ yêu cầu một toán hạng (ví dụ: ~, ++), và chỉ có một toán tử 3 ngôi duy nhất ?

## Toán tử quan hệ

Các toán tử quan hệ trong ngôn ngữ C kiểm tra xem một mối quan hệ cụ thể giữa hai toán hạng có đúng hay không. Kết quả được đánh giá là 1 (đúng) hoặc 0 (sai). Kết quả này thường được sử dụng để ảnh hưởng đến luồng điều khiển (thông qua câu lệnh if, while, for), nhưng cũng có thể được lưu trữ trong các biến.

### Bằng “==”

Kiểm tra xem các toán hạng được cung cấp có bằng nhau không.

|  |
| --- |
| 1 == 0; /\* đánh giá thành 0.(Sai) \*/  1 == 1; /\* đánh giá thành 1.(Đúng) \*/  int x = 5;  int y = 5;  int \*xptr = &x, yptr = &y;  xptr == yptr; /\* đánh giá thành 0, các toán hạng giữ các địa chỉ vùng nhớ khác nhau.(Sai) \*/  \*xptr == yptr; /\* đánh giá thành 1, các toán hạng trỏ tới các vị trí vùng nhớ chứa cùng một giá trị.(Đúng) \*/ |

**Chú ý:** Không nên nhầm lẫn toán tử này với toán tử gán “=” !!!

### Không bằng (Khác) “!=”

Kiểm tra xem các toán hạng được cung cấp có bằng nhau không

|  |
| --- |
| 1 == 0; /\* đánh giá thành 0.(Sai) \*/  1 == 1; /\* đánh giá thành 1.(Đúng) \*/  int x = 5;  int y = 5;  int \*xptr = &x, yptr = &y;  xptr == yptr; /\* đánh giá thành 0, các toán hạng giữ các địa chỉ vùng nhớ khác nhau.(Sai) \*/  \*xptr == yptr; /\* đánh giá thành 1, các toán hạng trỏ tới các vị trí vùng nhớ chứa cùng một giá trị.(Đúng) \*/ |

Toán tử này trả về kết quả ngược lại với kết quả của toán tử **bằng bằng** (==).

### Not “!”

Kiểm tra xem một đối tượng có bằng 0 hay không.

Toán tử "!" cũng có thể được sử dụng trực tiếp với một biến như sau:

|  |
| --- |
| !someVal |

Điểu này có tác dụng tương tự như

|  |
| --- |
| someVal == 0 |

### So sánh lớn hơn “>”

Kiểm tra xem toán hạng bên trái có giá trị lớn hơn toán hạng bên phải hay không

|  |
| --- |
| 5 > 4 /\* trả về 1.(Đúng) \*/  4 > 5 /\* trả về 0.(Sai) \*/   1. > 4 /\* trả về 0.(Sai) \*/ |

### So sánh bé hơn “<”

Kiểm tra xem toán hạng bên trái có giá trị bé hơn toán hạng bên phải hay không

|  |
| --- |
| 5 < 4 /\* trả về 0.(Sai) \*/  4 < 5 /\* trả về 1.(Đúng) \*/   1. < 4 /\* trả về 0.(Sai) \*/ |

### So sánh lớn hơn hoặc bằng “>=”

Kiểm tra xem toán hạng bên trái có giá trị lớn hơn hoặc bằng toán hạng bên phải hay không

|  |
| --- |
| 5 >= 4 /\* trả về 1.(Đúng) \*/  4 >= 5 /\* trả về 0.(Sai) \*/   1. >= 4 /\* trả về 1.(Đúng) \*/ |

### So sánh bé hơn hoặc bằng “<=”

Kiểm tra xem toán hạng bên trái có giá trị bé hơn hoặc bằng toán hạng bên phải hay không

|  |
| --- |
| 5 <= 4 /\* trả về 0.(Sai) \*/  4 <= 5 /\* trả về 1.(Đúng) \*/  4 <= 4 /\* trả về 1.(Đúng) \*/ |

## TOÁN TỬ ĐIỀU KIỆN/ TOÁN TỬ 3 NGÔI

Toán tử 3 ngôi "?" trong ngôn ngữ C được sử dụng để thực hiện một phép toán 3 ngôi (ternary operation). Nó đánh giá toán hạng đầu tiên và nếu giá trị kết quả khác 0, nó đánh giá toán hạng thứ hai. Ngược lại, nếu giá trị kết quả bằng 0, nó đánh giá toán hạng thứ ba.

|  |
| --- |
| a = b ? c : d;  /\*Tương đương với\*/  if (b)  a = c;  else  a = d; |

Đoạn mã sau đây mô tả cú pháp của toán tử 3 ngôi: **điều kiện ? kết quả nếu đúng : kết quả nếu sai**. Mỗi giá trị có thể là kết quả của một biểu thức đã được đánh giá.

|  |
| --- |
| int x = 5;  int y = 42;  printf("%i, %i\n", 1 ? x : y, 0 ? x : y); /\* Kết quả "5, 42" \*/ |

Toán tử điều kiện có thể được lồng vào nhau. Ví dụ: đoạn mã sau xác định số lớn hơn trong ba số:

|  |
| --- |
| big= a > b ? (a > c ? a : c)  : (b > c ? b : c); |

Ví dụ sau ghi số nguyên chẵn vào một tệp và số nguyên lẻ vào tệp khác

|  |
| --- |
| #include<stdio.h>  int main()  {  *FILE* \*even, \*odds;  int n = 10;  *size\_t* k = 0;  even = fopen("even.txt", "w");  odds = fopen("odds.txt", "w");  for(k = 1; k < n + 1; k++)  {  k%2==0 ? fprintf(even, "\t%5d\n", k)  : fprintf(odds, "\t%5d\n", k);  }  fclose(even);  fclose(odds);  return 0;  } |

Toán tử điều kiện liên kết từ phải sang trái. Hãy xem xét những điều sau đây:

|  |
| --- |
| exp1 ? exp2 : exp3 ? exp4 : exp5 |

Vì liên kết từ phải sang trái, biểu thức trên được đánh giá là:

|  |
| --- |
| exp1 ? exp2 : ( exp3 ? exp4 : exp5 ) |

## TOÁN TỬ BITWISE

Toán tử bitwise có thể được sử dụng để thực hiện thao tác cấp độ bit trên các biến. Dưới đây là danh sách tất cả sáu toán tử bitwise được hỗ trợ trong C:

|  |  |
| --- | --- |
| Kí hiệu | Toán tử |
| & | Toán tử AND |
| | | Toán tử OR |
| ^ | Toán tử XOR |
| ~ | Toán tử phủ định NOT |
| << | Toán tử dịch trái |
| >> | Toán tử dịch phải |

Chương trình sau đây minh họa việc sử dụng tất cả các toán tử bitwise:

|  |
| --- |
| #include <stdio.h>  int main()  {  unsigned int a = 29; /\* 29 = 0001 1101 \*/  unsigned int b = 48; /\* 48 = 0011 0000 \*/  int c = 0;  c = a & b; /\* 32 = 0001 0000 \*/  printf("%d & %d = %d\n", a, b, c );  c = a | b; /\* 61 = 0011 1101 \*/  printf("%d | %d = %d\n", a, b, c );  c = a ^ b; /\* 45 = 0010 1101 \*/  printf("%d ^ %d = %d\n", a, b, c );  c = ~a; /\* -30 = 1110 0010 \*/  printf("~%d = %d\n", a, c );  c = a << 2; /\* 116 = 0111 0100 \*/  printf("%d << 2 = %d\n", a, c );  c = a >> 2; /\* 7 = 0000 0111 \*/  printf("%d >> 2 = %d\n", a, c );  return 0;  } |

**Kết quả:**

|  |
| --- |
| **29 & 48 = 16**  **29 | 48 = 61**  **29 ^ 48 = 45**  **~29 = -30**  **29 << 2 = 116**  **29 >> 2 = 7** |

Các phép toán bitwise với kiểu dữ liệu có dấu nên được sử dụng cẩn thận vì bit dấu của kiểu dữ liệu đó có ý nghĩa riêng. Có các hạn chế đặc biệt áp dụng cho các toán tử dịch chuyển (shift operators):

* Dịch trái một bit 1 vào bit dấu của kiểu dữ liệu có dấu được coi là sai và dẫn đến hành vi không xác định. Điều này có nghĩa rằng kết quả của phép dịch như vậy không được xác định rõ ràng và có thể khác nhau tùy thuộc vào trình biên dịch và nền tảng.
* Dịch phải một giá trị âm (với bit dấu được đặt là 1) được xác định bởi từng trình biên dịch và do đó không có tính chất di động. Điều này có nghĩa rằng hành vi của phép dịch như vậy có thể khác nhau giữa các phiên bản khác nhau của ngôn ngữ C và không được đảm bảo di động trên các nền tảng hoặc trình biên dịch khác nhau.
* Nếu giá trị của toán hạng phải của toán tử dịch chuyển là số âm hoặc lớn hơn hoặc bằng độ rộng của toán hạng trái được gán, hành vi là không xác định. Điều này có nghĩa là kết quả của phép dịch như vậy không được chỉ định và có thể cho ra kết quả không mong đợi hoặc không nhất quán.

Để đảm bảo tính di động và tránh hành vi không xác định, thường khuyến nghị sử dụng kiểu dữ liệu không dấu khi thực hiện các phép toán bitwise liên quan đến dịch chuyển.

**Masking:**

Masking (hoặc còn gọi là áp dụng mặt nạ) đề cập đến quá trình trích xuất các bit mong muốn từ một biến (hoặc biến đổi các bit mong muốn trong biến) bằng cách sử dụng các phép toán logic bitwise. Toán hạng (một hằng số hoặc biến) được sử dụng để thực hiện việc áp dụng mặt nạ được gọi là mặt nạ (mask).

Trong quá trình masking, mặt nạ là một giá trị có các bit được thiết lập một cách đặc biệt để thực hiện việc trích xuất hoặc biến đổi các bit trong biến gốc. Khi thực hiện phép toán bitwise AND (&) giữa biến gốc và mặt nạ, chỉ có các bit tương ứng mà mặt nạ đặt thành 1 và các bit còn lại trong biến gốc sẽ bị xóa đi (thiết lập thành 0). Điều này cho phép trích xuất hoặc biến đổi các bit cần thiết trong biến gốc và bỏ qua các bit không mong muốn.

Mặt nạ được sử dụng theo nhiều cách khác nhau

* Để quyết định mẫu bit của một biến số nguyên.
* Để sao chép một phần của mẫu bit đã cho sang một biến mới, trong khi phần còn lại của biến mới được điền bằng 0 (sử dụng bitwise AND)
* Để sao chép một phần của mẫu bit đã cho sang một biến mới, trong khi phần còn lại của biến mới được lấp đầy bằng 1 (sử dụng OR theo bit).
* Để sao chép một phần của mẫu bit đã cho sang một biến mới, trong khi phần còn lại của mẫu bit ban đầu được đảo ngược trong biến mới (sử dụng OR loại trừ bit).

Hàm sau sử dụng mặt nạ để hiển thị mẫu bit của một biến:

|  |
| --- |
| #include <limits.h>  void bit\_pattern(int *u*)  {  int i, x, word;  unsigned mask = 1;  word = CHAR\_BIT \* sizeof(int);  mask = mask << (word - 1); /\* shift 1 to the leftmost position \*/  for(i = 1; i <= word; i++)  {  x = (*u* & mask) ? 1 : 0; /\* identify the bit \*/  printf("%d", x); /\* print bit value \*/  mask >>= 1; /\* shift mask to the right by 1 bit \*/  }  } |

## HOẠT ĐỘNG NGẮN MẠCH CỦA TOÁN TỬ LOGIC

## TOÁN TỬ DẤU PHẨY

Toán tử dấu phẩy (,) cho phép bạn đánh giá nhiều biểu thức ở bất kỳ nơi nào cho phép một biểu thức duy nhất. Toán tử dấu phẩy đánh giá toán hạng bên trái, sau đó là toán hạng bên phải, rồi trả về kết quả của toán hạng bên phải.

|  |
| --- |
| int x = 42, y = 42;  printf("%i\n", (x \*= 2, y)); /\* Outputs "42". \*/ |

Toán tử dấu phẩy giới thiệu một điểm thứ tự giữa các toán hạng của nó.

**Lưu ý** rằng dấu phẩy được sử dụng trong các hàm gọi các đối số riêng biệt đó **KHÔNG** phải là toán tử dấu phẩy, thay vào đó, nó được gọi là dấu phân cách khác với toán tử dấu phẩy. Do đó, nó không có các thuộc tính của toán tử dấu phẩy.

Lệnh gọi printf() ở trên chứa cả toán tử dấu phẩy và dấu phân cách.

|  |
| --- |
| printf("%i\n", (x \*= 2, y)); /\* Outputs "42". \*/  /\* ^ ^ Đây là toán tử dấu phẩy \*/  /\* Đây là dấu phân cách \*/ |

Toán tử dấu phẩy thường được sử dụng trong phần khởi tạo cũng như trong phần cập nhật của vòng lặp for. Ví dụ:

|  |
| --- |
| for(k = 1; k < 10; printf("*\%*d\\n", k), k += 2); /\*outputs the odd numbers below 9/\*  /\* outputs sum to first 9 natural numbers \*/  for(sumk = 1, k = 1; k < 10; k++, sumk += k)  printf("*\%*5d*\%*5d\\n", k, sumk); |

## TOÁN TỬ SỐ HỌC

### 4.6.1 Số học cơ bản

Trả về một giá trị là kết quả của việc áp dụng toán hạng bên trái cho toán hạng bên phải, sử dụng phép toán liên quan. Các quy tắc giao hoán toán học thông thường được áp dụng (nghĩa là phép cộng và phép nhân có tính chất giao hoán, phép trừ, phép chia và mô đun thì không).

### 4.6.2 Toán tử cộng

Toán tử cộng (+) được sử dụng để cộng hai toán hạng lại với nhau. Ví dụ:

|  |
| --- |
| #include<stdio.h>  int main(){  int a = 5;  int b = 7;  int c = a + b;  printf("%d + %d = %d",a,b,c);  return 0;  } |

**Kết quả**

|  |
| --- |
| **5 + 7 = 12** |

### 4.6.3 Toán tử trừ

Toán tử trừ được sử dụng để trừ toán hạng thứ 2 cho toán hạng thứ nhất. Ví dụ

|  |
| --- |
| #include<stdio.h>  int main(){  int a = 5;  int b = 7;  int c = a - b;  printf("%d - %d = %d",a,b,c);  return 0;  } |

**Kết quả**

|  |
| --- |
| **5 - 7 = -2** |

### 4.6.4 Toán tử nhân

Toán tử nhân (\*) được sử dụng để nhân cả hai toán hạng. Ví dụ:

|  |
| --- |
| #include<stdio.h>  int main(){  int a = 5;  int b = 7;  int c = a \* b;  printf("%d \* %d = %d",a,b,c);  return 0;  } |

**Kết quả**

|  |
| --- |
| **5 \* 7 = 35** |

**Đừng nhầm lẫn với con trỏ (\*)**

### 4.6.5 Toán tử chia

Toán tử chia (/) chia toán hạng thứ nhất cho toán hạng thứ hai. Nếu cả hai toán hạng của phép chia là số nguyên, nó sẽ trả về một giá trị nguyên và loại bỏ phần còn lại (sử dụng toán tử modulo % để tính toán và lấy phần còn lại).

Nếu một trong các toán hạng là một giá trị dấu chấm động, thì kết quả là một phép tính gần đúng của phân số.

**Ví dụ:**

|  |
| --- |
| #include<stdio.h>  int main(){  int a = 19 / 2 ;  int b = 18 / 2 ;  int c = 255 / 2;  int d = 44 / 4 ;  double e = 19 / 2.0 ;  double f = 18.0 / 2 ;  double g = 255 / 2.0;  double h = 45.0 / 4 ;  printf("19 / 2 = %d\n", a);  printf("18 / 2 = %d\n", b);  printf("255 / 2 = %d\n", c);  printf("44 / 4 = %d\n", d);  printf("19 / 2.0 = %g\n", e);  printf("18.0 / 2 = %g\n", f);  printf("255 / 2.0 = %g\n", g);  printf("45.0 / 4 = %g\n", h);  } |

**Kết quả**

|  |
| --- |
| **19 / 2 = 9**  **18 / 2 = 9**  **255 / 2 = 127**  **44 / 4 = 11**  **19 / 2.0 = 9.5**  **18.0 / 2 = 9**  **255 / 2.0 = 127.5**  **45.0 / 4 = 11.25** |

**Chú ý: Ép kiểu**

|  |
| --- |
| #include<stdio.h>  int main(){  int a = 9;  int b = 2;  float c;  //c = (float) a / b;  //c = 9.0 / 2;  c = 9 / 2.0;  printf("%d / %d = %f",a,b,c);  } |

**Kết quả**

|  |
| --- |
| **9 / 2 = 4.500000** |

### 4.6.6 Toán tử modulo (Phép chia lấy dư)

Toán tử modulo (%) chỉ nhận toán hạng số nguyên và được sử dụng để tính phần còn lại sau khi toán hạng thứ nhất được chia cho toán hạng thứ hai. Toán tử mô đun (còn được gọi là toán tử chia lấy phần dư) là một toán tử trả về phần còn lại sau khi thực hiện phép chia số nguyên.

Ví dụ:

|  |
| --- |
| #include <stdio.h>  int main (void) {  int a = 25 % 2; /\* a holds value 1 \*/  int b = 24 % 2; /\* b holds value 0 \*/  int c = 155 % 5; /\* c holds value 0 \*/  int d = 49 % 25; /\* d holds value 24 \*/  printf("25 *%* 2 = %d\n", a); /\* Will output "25 % 2 = 1" \*/  printf("24 *%* 2 = %d\n", b); /\* Will output "24 % 2 = 0" \*/  printf("155 *%* 5 = %d\n", c); /\* Will output "155 % 5 = 0" \*/  printf("49 *%* 25 = %d\n", d); /\* Will output "49 % 25 = 24" \*/  return 0;  } |

### 4.6.7 Toán tử tăng giảm

Các toán tử tăng (a++) và giảm (a--) khác nhau ở chỗ chúng thay đổi giá trị của biến mà bạn áp dụng chúng mà không có toán tử gán. Bạn có thể sử dụng toán tử **tăng** và **giảm** trước hoặc sau biến. Vị trí của toán tử thay đổi thời gian tăng/giảm giá trị thành trước hoặc sau khi gán nó cho biến.

Ví dụ:

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int a = 1;  int b = 4;  int c = 1;  int d = 4;  a++;  printf("a = %d\n",a); /\* Sẽ in ra "a = 2" \*/  b--;  printf("b = %d\n",b); /\* Sẽ in ra "b = 3" \*/  if (++c > 1) { /\* c được tăng thêm 1 trước khi được so sánh trong điều kiện \*/  printf("This will print\n"); /\* Dòng này sẽ được in ra \*/  } else {  printf("This will never print\n"); /\* Dòng này sẽ không được in ra \*/  }  if (d-- < 4) { /\* d được giảm sau khi được so sánh \*/  printf("This will never print\n"); /\* Dòng này sẽ không được in ra \*/  } else {  printf("This will print\n"); /\* Dòng này sẽ được in ra \*/  }  } |

Kết quả

|  |
| --- |
| a = 2  b = 3  This will print  This will print |

Như ví dụ cho c và d cho thấy, cả hai toán tử đều có hai dạng, là ký hiệu tiền tố và ký hiệu hậu tố. Cả hai đều có tác dụng giống nhau trong việc tăng (++) hoặc giảm (--) biến, nhưng khác nhau về giá trị mà chúng trả về: các phép toán tiền tố thực hiện thao tác trước rồi trả về giá trị, trong khi các phép toán hậu tố trước tiên xác định giá trị được trả lại, và sau đó thực hiện thao tác.

Do hành vi có khả năng phản trực giác này, việc sử dụng các toán tử tăng/giảm bên trong các biểu thức đang gây tranh cãi.

## TOÁN TỬ TRUY CẬP

Các toán tử truy cập thành viên (dấu chấm . và mũi tên ->) được sử dụng để truy cập một thành viên của cấu trúc(struct).

### 4.7.1 Thuộc tính của đối tượng

Được đánh giá thành giá trị, chỉ định đối tượng là thành viên của đối tượng được truy cập.

|  |
| --- |
| struct *MyStruct*  {  int x;  int y;  };  struct *MyStruct* myObject;  myObject.x = 42;  myObject.y = 123;  printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /\* Outputs ".x = 42, .y = 123". \*/ |

### 4.7.2 Thành viên của đối tượng được trỏ tới

Toán tử mũi tên (->) được sử dụng để tiện lợi hóa quá trình giải tham chiếu và truy cập thành viên của một cấu trúc. Thay vì viết (\*x).y để truy cập thành viên y của đối tượng được giải tham chiếu bởi con trỏ x, chúng ta có thể sử dụng x->y. Điều này giúp làm rõ hơn cách thao tác và làm cho mã nguồn dễ đọc hơn, đặc biệt là khi có các con trỏ cấu trúc lồng nhau.

|  |
| --- |
| #include <stdio.h>  int main(){  struct *MyStruct*  {  int x;  int y;  };  struct *MyStruct* myObject;  struct *MyStruct* \*p = &myObject;  p->x = 42;  p->y = 123;  printf(".x = %i, .y = %i\n", p->x, p->y); /\* Kết quả ".x = 42, .y = 123". \*/  printf(".x = %i, .y = %i\n", myObject.x, myObject.y); /\* Kết quả cũng sẽ là ".x = 42, .y = 123". \*/  } |

### 4.7.3 Lấy địa chỉ

Toán tử & (unary address of operator) trong ngôn ngữ C được sử dụng để lấy địa chỉ của một biến. Khi áp dụng toán tử & cho một biến, nó trả về một con trỏ chứa địa chỉ của biến đó.

**Ví dụ:**

|  |
| --- |
| #include <stdio.h>  int main(){  int x = 3;  int \*p = &x;  printf("%p = %p\n", &x, (p)); /\* Xuất ra "A = A",Với A là một giá trị do triển khai cụ thể quy định ở đây là giá trị địa chỉ ngẫu nhiên.\*/  } |

**Kết quả**

|  |
| --- |
| 0061FF18 = 0061FF18 |

### 4.7.4 Dereference

The unary \* operator dereferences a pointer. It evaluates into the lvalue resulting from dereferencing the pointer that results from evaluating the given expression.

|  |
| --- |
| int x = 42;  int \*p = &x;  printf("x = %d, \*p = %d\n", x, \**p*); /\* Xuất ra "x = 42, \*p = 42". \*/  \*p = 123;  printf("x = %d, \*p = %d\n", x, \**p*); /\* Xuất ra "x = 123, \*p = 123". \*/ |

### 4.7.5 Chỉ số mảng

Trong C, cú pháp indexing (chỉ số mảng) thực tế là một cách viết tắt cho việc thực hiện phép cộng con trỏ và sau đó giải tham chiếu. Biểu thức dạng **a[i]** tương đương với **\*(a + i)**, trong đó a là một con trỏ và i là chỉ số hoặc độ lệch.

Trong C, mảng sẽ tự động chuyển thành con trỏ tới phần tử đầu tiên của nó trong hầu hết các ngữ cảnh. Khi bạn sử dụng cú pháp chỉ số **a[i],** trình biên dịch sẽ coi a như một con trỏ tới phần tử đầu tiên của mảng và thực hiện phép cộng con trỏ với độ lệch **i**. Sau đó, con trỏ kết quả được giải tham chiếu sử dụng toán tử **\*** để truy cập vào giá trị tại vị trí đó trong bộ nhớ.

|  |
| --- |
| int arr[] = { 1, 2, 3, 4, 5 };  printf("arr[2] = %i\n", arr[2]); /\* Xuất ra "arr[2] = 3". \*/ |

### 4.7.6 Khả năng hoán đổi của chỉ số mảng

Thêm một con trỏ vào một số nguyên là một phép toán giao hoán (nghĩa là thứ tự của các toán hạng không làm thay đổi kết quả) vì vậy con trỏ + số nguyên == số nguyên + con trỏ.

Hệ quả của điều này là arr[3] và 3[arr] là tương đương.

|  |
| --- |
| printf("3[arr] = %i\n", 3[arr]); /\* Xuất ra "3[arr] = 4". \*/ |

## PHẦN 4.8 TOÁN TỬ SIZEOF

### 4.8.1 With a type as operand

Đánh giá kết quả là kích thước trong byte, của kiểu size\_t, của đối tượng thuộc kiểu đã cho. Yêu cầu có dấu ngoặc đơn xung quanh kiểu dữ liệu.

|  |
| --- |
| printf("%zu\n", sizeof(int)); /\* Hợp lệ, in ra kích thước của một đối tượng int,  \*có thể thay đổi theo nền tảng. \*/  printf("%zu\n", sizeof *int*); /\* Không hợp lệ, các kiểu dữ liệu truyền vào cần được  \*bao bọc bởi dấu ngoặc đơn! \*/ |

### 4.8.2 With an expression as operand

Đánh giá kết quả là kích thước trong byte, kiểu size\_t, của các đối tượng có cùng kiểu dữ liệu với biểu thức được cung cấp. Biểu thức chính nó không được đánh giá. Dấu ngoặc đơn không bắt buộc; tuy nhiên, vì biểu thức được cung cấp phải là toán tử một ngôi, nên luôn tốt nhất là sử dụng chúng.

|  |
| --- |
| char ch = 'a';  printf("%zu\n", sizeof(ch)); /\* Hợp lệ, sẽ in ra kích thước của một đối tượng char,  \* luôn là 1 cho mọi nền tảng. \*/  printf("%zu\n", sizeof *ch*); /\* Hợp lệ, sẽ in ra kích thước của một đối tượng char,  \*luôn là 1 cho mọi nền tảng. \*/ |

## TOÁN TỬ ÉP KIỂU

Thực hiện một chuyển đổi rõ ràng sang kiểu dữ liệu đã chỉ định từ giá trị thu được sau khi đánh giá biểu thức cung cấp.

|  |
| --- |
| int x = 3;  int y = 4;  printf("%f\n", (double)x / y); /\* In ra "0.750000". \*/ |

Ở đây, giá trị của x được chuyển đổi thành kiểu double, phép chia cũng chuyển đổi giá trị của y thành kiểu double và kết quả của phép chia, một giá trị double, được truyền vào hàm printf để in ra màn hình.

## TOÁN TỬ GỌI HÀM

Toán tử gọi hàm trong ngôn ngữ lập trình yêu cầu toán hạng đầu tiên phải là một con trỏ tới hàm (function pointer) hoặc một biểu thức chỉ định hàm (function designator). Đối tượng này xác định hàm mà ta muốn gọi. Các toán hạng còn lại, nếu có, được gọi là đối số của cuộc gọi hàm. Toán tử này đánh giá thành giá trị trả về từ việc gọi hàm thích hợp với các đối số tương ứng.

|  |
| --- |
| int myFunction(int *x*, int *y*)  {  return *x* \* 2 + *y*;  }  int (\*fn)(int, int) = &myFunction;  int x = 42;  int y = 123;  printf("(\*fn)(%i, %i) = %i\n", x, y, (fn)(x, y)); /\* In ra "fn(42, 123) = 207". \*/  printf("fn(%i, %i) = %i\n", x, y, fn(*x*, y)); /\* Một dạng khác: bạn không cần phải giải tham chiếu một cách rõ ràng \*/ |

## TOÁN TỬ TĂNG/ TOÁN TỬ GIẢM

Các toán tử tăng và giảm tồn tại ở dạng tiền tố và hậu tố.

|  |
| --- |
| int a = 1;  int b = 1;  int tmp = 0;  tmp = ++a; /\* tăng a lên một đơn vị và trả về giá trị mới; a == 2, tmp == 2 \*/  tmp = a++; /\* tăng a lên một đơn vị nhưng trả về giá trị cũ; a == 3, tmp == 2 \*/  tmp = --b; /\* giảm b đi một đơn vị và trả về giá trị mới; b == 0, tmp == 0 \*/  tmp = b--; /\* giảm b đi một đơn vị nhưng trả về giá trị cũ; b == -1, tmp == 0 \*/ |

Hiểu đơn giản

Toán tử ++: Tăng giá trị lên 1 đơn vị.

**• x++:** thực hiện lệnh trước rồi mới tăng x lên 1 đơn vị.

**• ++x:** tăng x lên 1 đơn vị rồi mới thực hiện lệnh.

Toán tử --: Giảm giá trị đi 1 đơn vị.

**• x--:** thực hiện lệnh trước rồi mới giảm x đi 1 đơn vị.

**• --x:** giảm x đi 1 đơn vị rồi mới thực hiện lệnh.

Lưu ý rằng các phép toán số học không tạo ra điểm chuỗi (sequence points), do đó, một số biểu thức sử dụng toán tử tăng (++) hoặc giảm (--) có thể dẫn đến hành vi không xác định.

## TOÁN TỬ GÁN

Gán giá trị của toán hạng bên phải cho vị trí lưu trữ được đặt tên bởi toán hạng bên trái và trả về giá trị.

|  |
| --- |
| int x = 5; /\* Biến x chứa giá trị 5. Trả về 5. /  char y = 'c'; /\* Biến y chứa giá trị 99. Trả về 99  \*(vì ký tự 'c' được đại diện bởi 99 trong bảng ASCII). \*/  float z = 1.5; /\* Biến z chứa giá trị 1.5. Trả về 1.5. \*/  char const\* s = "foo"; /\* Biến s chứa địa chỉ của ký tự đầu tiên trong chuỗi 'foo'. \*/ |

Một số phép tính số học có toán tử gán phức hợp.

|  |
| --- |
| a += b /\* tương đương với: a = a + b \*/  a -= b /\* tương đương với: a = a - b \*/  a = b /\* tương đương với: a = a \* b \*/  a /= b /\* tương đương với: a = a / b \*/  a %= b /\*tương đương với: a = a % b \*/  a &= b /\* tương đương với: a = a & b \*/  a |= b /\* tương đương với: a = a | b \*/  a ^= b /\* tương đương với: a = a ^ b \*/  a <<= b /\* tương đương với: a = a << b \*/  a >>= b /\*tương đương với: a = a >> b \*/ |

Một tính năng quan trọng của các toán tử gán kết hợp này là biểu thức ở phía bên trái (a) chỉ được đánh giá một lần. Ví dụ, nếu p là một con trỏ:

|  |
| --- |
| \*p += 5; |

Trong trường hợp này, giá trị của biểu thức \*p sẽ chỉ được tính toán một lần và sau đó, giá trị mới được gán vào vị trí mà con trỏ p trỏ tới. Điều này giúp tránh các lỗi không xác định hoặc không mong muốn khi giá trị của biểu thức thay đổi trong quá trình đánh giá.

|  |
| --- |
| \*p = \*p + 5 |

Cũng cần lưu ý rằng kết quả của một phép gán chẳng hạn như a = b là cái được gọi là giá trị. Do đó, phép gán thực sự có một giá trị mà sau đó có thể được gán cho một biến khác. Điều này cho phép xâu chuỗi các phép gán để đặt nhiều biến trong một câu lệnh.

Giá trị này có thể được sử dụng trong các biểu thức kiểm soát của câu lệnh if (hoặc vòng lặp hoặc câu lệnh chuyển đổi) để bảo vệ một số mã trên kết quả của một biểu thức hoặc lời gọi hàm khác. Ví dụ:

|  |
| --- |
| char \*buffer;  if ((buffer = malloc(1024)) != NULL)  {  /\* thực hiện một số công việc sử dụng buffer \*/  free(buffer);  }  else  {  /\* báo cáo lỗi khi không cấp phát được bộ nhớ \*/  } |

Do đó, cần phải cẩn thận để tránh mắc lỗi đánh máy phổ biến có thể dẫn đến các lỗi bí ẩn.

|  |
| --- |
| int a = 2;  /\* ... \*/  if (a = 1)  /\* Xóa tất cả các tập tin trên ổ cứng của tôi \*/ |

Điều này sẽ dẫn đến kết quả tai hại, vì a = 1 sẽ luôn đánh giá bằng 1 và do đó, biểu thức kiểm soát của câu lệnh if sẽ luôn đúng (đọc thêm về cạm bẫy phổ biến này tại đây). Tác giả gần như chắc chắn có ý định sử dụng toán tử đẳng thức (==) như hình dưới đây:

|  |
| --- |
| int a = 2;  /\* ... \*/  if (a == 1)  /\* Xóa tất cả các tập tin trên ổ cứng của tôi \*/ |

**Tính kết hợp của toán tử**

|  |
| --- |
| int a, b = 1, c = 2;  a = b = c; |

Điều này gán c cho b, trả về b, được gán cho a. Điều này xảy ra bởi vì tất cả các toán tử gán có tính kết hợp phải, điều đó có nghĩa là phép toán ngoài cùng bên phải trong biểu thức được ước tính trước và tiến hành từ phải sang trái.

## TOÁN TỬ LOGIC

### 4.13.1 Logic AND

Toán tử logic AND (&&) trong C thực hiện phép AND logic giữa hai toán hạng và trả về giá trị 1 nếu cả hai toán hạng đều khác 0. (Có nghĩa là đúng khi cả 2 đều đúng) Toán tử AND logic có kiểu int.

|  |
| --- |
| 0 && 0 /\* Trả về 0. \*/  0 && 1 /\* Trả về 0. \*/  2 && 0 /\* Trả về 0. \*/  2 && 3 /\* Trả về 1. \*/ |

### 4.13.2 Logic OR

Toán tử logic OR (||) trong C thực hiện phép OR logic giữa hai toán hạng và trả về giá trị 1 nếu bất kỳ một trong hai toán hạng là khác không. Toán tử OR logic có kiểu int.

|  |
| --- |
| 0 || 0 /\* Trả về 0. \*/  0 || 1 /\* Trả về 1. \*/  2 || 0 /\* Trả về 1. \*/  2 || 3 /\* Trả về 1. \*/ |

### 4.13.3 Logic NOT

Toán tử logic NOT (!) trong C thực hiện phép phủ định logic của một biểu thức và trả về giá trị 1 nếu biểu thức là sai (bằng 0), và trả về giá trị 0 nếu biểu thức là đúng (khác 0). Toán tử NOT logic có kiểu int.

Các tính chất quan trọng chung cho cả phép toán logic AND (&&) và phép toán logic OR (||) là như sau, và rất quan trọng để hiểu cách chúng hoạt động trong việc đánh giá ngắn gọn (short-circuit evaluation):

Toán hạng bên trái (LHS) được đánh giá hoàn toàn trước khi toán hạng bên phải (RHS) được đánh giá:

* Trong biểu thức A && B, A được đánh giá trước, và chỉ khi A là đúng (khác không) thì B mới được đánh giá. Nếu A là sai (bằng không), B sẽ không được đánh giá vì kết quả tổng thể của phép toán AND đã xác định là sai.
* Trong biểu thức A || B, A được đánh giá trước, và chỉ khi A là sai (bằng không) thì B mới được đánh giá. Nếu A là đúng (khác không), B sẽ không được đánh giá vì kết quả tổng thể của phép toán OR đã xác định là đúng.

Có một điểm chuỗi (sequence point) giữa việc đánh giá toán hạng bên trái và toán hạng bên phải:

* Điểm chuỗi là điểm trong việc thực thi chương trình nơi tất cả các hiệu ứng phụ của các đánh giá trước đó đã được thực hiện. Điểm chuỗi đảm bảo rằng các đánh giá xảy ra theo một thứ tự được xác định, tránh các vấn đề liên quan đến hành vi không xác định (undefined behavior).

Toán hạng bên phải không được đánh giá nếu kết quả của toán hạng bên trái xác định kết quả tổng thể:

* Đánh giá ngắn gọn (short-circuit evaluation) tránh việc thực hiện các tính toán không cần thiết. Ví dụ, trong A && B, nếu A là sai, kết quả của biểu thức tổng thể đã biết là sai bất kể giá trị của B. Do đó, không cần phải đánh giá B.
* Tương tự, trong A || B, nếu A là đúng, kết quả của biểu thức tổng thể đã biết là đúng bất kể giá trị của B. Do đó, không cần phải đánh giá B.

Các tính chất này là rất quan trọng để viết mã hiệu quả và an toàn, đặc biệt khi sử dụng các biểu thức có hiệu ứng phụ tiềm ẩn hoặc các đánh giá phức tạp. Nó cho phép các lập trình viên tối ưu hóa mã của họ và tránh công việc không cần thiết khi đánh giá biểu thức logic.

## TOÁN HỌC TRÊN CON TRỎ

### 4.14.1 Cộng con trỏ

### 4.14.2 Trừ con trỏ

## \_Alignof

# BOOLEAN

## PHẦN 5.1 SỬ DỤNG STDBOOL.H

Tiêu sử dụng tệp tiêu đề hệ thống **stdbool.h** cho phép bạn sử dụng **bool** như một kiểu dữ liệu Boolean. **true** tương đương với **1** và **false** tương đương với **0**.

|  |
| --- |
| #include <stdio.h>  #include <stdbool.h>  int main(void) {  bool x = true; /\* tương đương với bool x = 1; \*/  bool y = false; /\* tương đương với bool y = 0; \*/  if (x) /\* Tương đương với if (x != 0) hoặc if (x != false) \*/  {  puts("This will print!"); /\* In ra màn hình chuỗi "This will print!" \*/  }  if (!y) /\* Tương đương với if (y == 0) hoặc if (y == false) \*/  {  puts("This will also print!"); /\* In ra màn hình chuỗi "This will also print!" \*/  }  return 0;  } |

Trong C, từ khóa "bool" chỉ là một tên gọi khác cho kiểu dữ liệu "\_Bool". Kiểu dữ liệu "\_Bool" là một kiểu dữ liệu tích hợp được giới thiệu trong tiêu chuẩn C99 để biểu diễn giá trị boolean, chỉ có hai giá trị có thể xảy ra: đúng (1) và sai (0).

Kiểu dữ liệu "\_Bool" có các quy tắc đặc biệt khi chuyển đổi các số hoặc con trỏ sang nó.

## PHẦN 5.2 SỬ DỤNG #DEFINE

Trong C, mọi phiên bản, các toán tử so sánh thực sự sẽ xem xét bất kỳ giá trị số nguyên nào khác 0 là đúng (true) và giá trị số nguyên 0 là sai (false). Điều này áp dụng cho các biểu thức điều kiện như trong câu lệnh if, while, for và các phép so sánh khác.

Nếu bạn không có sẵn \_Bool hoặc bool như trong C99, bạn có thể mô phỏng một kiểu dữ liệu Boolean trong C bằng cách sử dụng các macro được định nghĩa bằng #define, và bạn có thể vẫn tìm thấy những điều tương tự trong mã nguồn cũ.

|  |
| --- |
| #include <stdio.h>  #define bool int  #define true 1  #define false 0  int main(void) {  bool x = true; /\* Tương đương với int x = 1; \*/  bool y = false; /\* Tương đương với int y = 0; \*/  if (x) /\* Tương đương với if (x != 0) hoặc if (x != false) \*/  {  puts("This will print!"); /\* In ra màn hình chuỗi "This will print!" \*/  }  if (!y) /\* Tương đương với if (y == 0) hoặc if (y == false) \*/  {  puts("This will also print!"); /\* In ra màn hình chuỗi "This will also print!" \*/  }  return 0;  } |

Trong mã nguồn mới, nên tránh định nghĩa những macro như bool, true, và false, vì chúng có thể gây xung đột với việc sử dụng hiện đại của thư viện <stdbool.h>.

## PHẦN 5.3 SỬ DỤNG KIỂU \_BOOL NỘI TẠI(TÍCH HỢP)

Kiểu dữ liệu \_Bool được thêm vào trong tiêu chuẩn C99 và là một kiểu dữ liệu nguyên thuỷ (native) trong ngôn ngữ C. Nó có khả năng lưu trữ các giá trị **0** (để biểu diễn **false**) và **1** (để biểu diễn **true**).

|  |
| --- |
| #include <stdio.h>  int main(void) {  \_Bool x = 1; /\* \_Bool có thể lưu trữ giá trị 0 hoặc 1 \*/  \_Bool y = 0;  if (x) /\* Tương đương với if (x == 1) \*/  {  puts("This will print!"); /\* In ra màn hình chuỗi "This will print!" \*/  }  if (!y) /\* Tương đương với if (y == 0) \*/  {  puts("This will also print!"); /\* In ra màn hình chuỗi "This will also print!" \*/  }  return 0;  } |

Trong ngôn ngữ C, \_Bool là một kiểu số nguyên, nhưng nó có các quy tắc đặc biệt cho việc chuyển đổi từ các kiểu dữ liệu khác khi được sử dụng trong biểu thức. Kết quả của các biểu thức \_Bool tương tự như việc sử dụng các kiểu số nguyên khác trong các biểu thức if. Trong ví dụ sau:

|  |
| --- |
| \_Bool z = X; |

Nếu X có kiểu số học (là bất kỳ loại số nào), thì z sẽ trở thành 0 nếu X bằng 0. Ngược lại, z sẽ trở thành 1.

Nếu X có kiểu con trỏ (kiểu dữ liệu liên quan đến con trỏ), thì z sẽ trở thành 0 nếu X là một con trỏ null (con trỏ không trỏ tới địa chỉ nào). Ngược lại, z sẽ trở thành 1 nếu X là một con trỏ hợp lệ (con trỏ trỏ tới địa chỉ nào đó).

Để sử dụng cách viết đẹp hơn với từ khóa bool, false, và true, bạn nên bao gồm tiêu đề <stdbool.h> trong mã C của mình.

## PHẦN 5.4 SỐ NGUYÊN VÀ CON TRỎ TRONG BIỂU THỨC BOOLEAN

Trong ngôn ngữ C, tất cả các số nguyên hoặc con trỏ đều có thể được sử dụng trong các biểu thức mà được hiểu là "giá trị đúng" (truth value) trong ngữ cảnh Logic.

|  |
| --- |
| int main(int *argc*, char\* *argv*[]) {  if (*argc* % 4) {  puts("arguments number is not divisible by 4");  } else {  puts("argument number is divisible by 4");  }  ... |

Biểu thức argc % 4 sẽ đánh giá thành một trong các giá trị 0, 1, 2 hoặc 3, tùy thuộc vào phần dư khi argc chia cho 4

* Nếu kết quả của argc % 4 là 0, nó sẽ được coi là "sai" (false) trong ngữ cảnh Logic (Boolean), và chương trình sẽ thực thi mã trong khối else, in ra thông báo "argument number is divisible by 4" (số lượng đối số chia hết cho 4).
* Nếu kết quả của argc % 4 là 1, 2 hoặc 3, nó sẽ được coi là "đúng" (true) trong ngữ cảnh Logic (Boolean), và chương trình sẽ thực thi mã trong khối if, in ra thông báo "arguments number is not divisible by 4" (số lượng đối số không chia hết cho 4).

|  |
| --- |
| double\* A = malloc(n \* sizeof(\*A));  if ( !A ) {  perror("allocation problems");  exit(EXIT\_FAILURE);  } |

Trong đoạn mã trên, ta cấp phát bộ nhớ động cho con trỏ A bằng hàm malloc và sau đó kiểm tra xem A có là con trỏ null hay không. Nếu A là con trỏ null, đoạn mã in ra thông báo lỗi và chương trình thoát. Điều này giúp phát hiện và xử lý vấn đề về cấp phát bộ nhớ một cách chính xác.

|  |
| --- |
| char const\* s = ....; /\* some pointer that we receive \*/  if (s != NULL && s[0] != '\0' && isalpha(s[0])) {  printf("this starts well, %c is alphabetic\n", s[0]);  } |

Để kiểm tra điều này, bạn phải quét mã phức tạp trong biểu thức và chắc chắn về tùy chọn toán tử.

|  |
| --- |
| char const\* s = ....; /\* some pointer that we receive \*/  if (s && s[0] && isalpha(s[0])) {  printf("this starts well, %c is alphabetic\n", s[0]);  } |

tương đối dễ nắm bắt: nếu con trỏ hợp lệ, chúng tôi kiểm tra xem ký tự đầu tiên có khác 0 hay không và sau đó kiểm tra xem đó có phải là một chữ cái hay không.

## PHẦN 5.5 XÁC ĐỊNH KIỂU BOOL BẰNG CÁCH SỬ DỤNG TYPEDEF

Thay vì sử dụng các **#define** macros, ta có thể sử dụng **enum** để định nghĩa các hằng số có tên để biểu thị giá trị true và false.

|  |
| --- |
| #if \_\_STDC\_VERSION\_\_ < 199900L  typedef enum { false, true } bool;  /\* Modern C code might expect these to be macros. \*/  # ifndef bool  # define bool bool  # endif  # ifndef true  # define true true  # endif  # ifndef false  # define false false  # endif  #else  # include <stdbool.h>  #endif  /\* Somewhere later in the code ... \*/  bool b = true; |

Điều này cho phép trình biên dịch cho các phiên bản C lịch sử hoạt động, nhưng vẫn tương thích về phía trước nếu mã được biên dịch bằng trình biên dịch C hiện đại.

Để biết thêm thông tin về typedef, hãy xem Typedef, để biết thêm về enum, hãy xem Enumerations

# CHUỖI

Trong ngôn ngữ C, chuỗi (string) không phải là một kiểu dữ liệu tự nhiên như số nguyên hay số thực. Thay vào đó, một chuỗi C được biểu diễn dưới dạng một mảng một chiều các ký tự (character array), kết thúc bằng ký tự null (null-character) là '\0'.

Điều này có nghĩa là một chuỗi C có nội dung là "abc" sẽ có bốn ký tự 'a', 'b', 'c' và '\0'.

Xem ví dụ giới thiệu cơ bản về chuỗi.

## PHẦN 6.1 PHÂN TÁCH TỪ: STRTOK(), STRTOK\_R() VÀ STRTOK\_S()

Hàm **strtok()** trong ngôn ngữ C được sử dụng để chia một chuỗi thành các chuỗi nhỏ hơn hoặc các phần tử con gọi là "**tokens**" bằng cách sử dụng một tập hợp các ký tự phân tách.

**Ví dụ:**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void)  {  int toknum = 0;  char src[] = "Hello,, world!";  const char delimiters[] = ", !";  char \*token = strtok(src, delimiters);  while (token != NULL)  {  printf("%d: [%s]\n", ++toknum, token);  token = strtok(NULL, delimiters);  }  /\* source is now "Hello\0, world\0\0" \*/  } |

**Kết quả:**

|  |
| --- |
| 1: [Hello]  2: [world] |

Chuỗi ký tự phân tách có thể chứa một hoặc nhiều ký tự phân tách, và bạn có thể sử dụng các chuỗi phân tách khác nhau trong mỗi lần gọi strtok().

Để tiếp tục phân tách chuỗi nguồn mà không truyền lại chuỗi nguồn, bạn nên truyền NULL làm đối số đầu tiên. Nếu truyền lại chuỗi nguồn, lần gọi đầu tiên của **strtok()** sẽ trả về lại token đầu tiên.

Hàm **strtok()** không cấp phát bộ nhớ mới cho các token, mà thay vào đó nó sửa đổi chuỗi nguồn. Điều này có nghĩa là chuỗi nguồn không thể là **const** (không thể là một con trỏ tới chuỗi cố định). Nó cũng có nghĩa là danh tính của ký tự phân tách bị mất (ví dụ: trong ví dụ trên, các dấu , và ! đã bị xóa khỏi chuỗi nguồn và không thể xác định được ký tự phân tách nào được trùng khớp).

Nếu có nhiều ký tự phân tách liên tiếp trong chuỗi nguồn, chúng sẽ được xem như một ký tự phân tách duy nhất; trong ví dụ, dấu phẩy thứ hai sẽ bị bỏ qua.

strtok() không an toàn khi sử dụng đa luồng hoặc đa ngõ ra do sử dụng **bộ đệm tĩnh** trong quá trình phân tích. Điều này có nghĩa là nếu một hàm gọi strtok(), bất kỳ hàm nào mà nó gọi trong khi sử dụng strtok() cũng không thể sử dụng strtok(), và nó không thể được gọi bởi bất kỳ hàm nào đang sử dụng strtok().

Dưới đây là một ví dụ minh họa cho vấn đề do việc strtok() không đảm bảo tính re-entrant (không đảm bảo khả năng sử dụng lại trong đa luồng). Trong môi trường đa luồng, việc gọi strtok() từ nhiều luồng có thể dẫn đến hành vi không mong muốn và làm mất dữ liệu.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void)  {  char src[] = "1.2,3.5,4.2";  char \*first = strtok(src, ",");  do  {  char \*part;  /\* Nested calls to strtok do not work as desired \*/  printf("[%s]\n", first);  part = strtok(first, ".");  while (part != NULL)  {  printf(" [%s]\n", part);  part = strtok(NULL, ".");  }  } while ((first = strtok(NULL, ",")) != NULL);  } |

**Kết quả:**

|  |
| --- |
| [1.2]  [1]  [2] |

Hoạt động dự kiến là vòng lặp do while bên ngoài sẽ tạo ba mã thông báo bao gồm mỗi chuỗi số thập phân ("1.2", "3.5", "4.2"), đối với mỗi mã mà strtok gọi cho vòng lặp bên trong sẽ chia nó thành các chuỗi riêng biệt chuỗi chữ số ("1", "2", "3", "5", "4", "2").

Tuy nhiên, vì strtok không đăng ký lại nên điều này không xảy ra. Thay vào đó, strtok đầu tiên tạo chính xác mã thông báo "1.2\0" và vòng lặp bên trong tạo chính xác mã thông báo "1" và "2". Nhưng sau đó, strtok trong vòng lặp bên ngoài nằm ở cuối chuỗi được sử dụng bởi vòng lặp bên trong và trả về NULL ngay lập tức. Chuỗi con thứ hai và thứ ba của mảng src hoàn toàn không được phân tích.

Các thư viện C tiêu chuẩn không chứa phiên bản an toàn cho luồng hoặc đăng ký lại nhưng một số phiên bản khác thì có, chẳng hạn như POSIX' strtok\_r. Lưu ý rằng trên MSVC, strtok tương đương, strtok\_s an toàn cho luồng.

C11 có một phần tùy chọn, Phụ lục K, cung cấp phiên bản đăng ký lại và an toàn cho luồng có tên strtok\_s. Bạn có thể kiểm tra tính năng này bằng \_\_STDC\_LIB\_EXT1\_\_. Phần tùy chọn này không được hỗ trợ rộng rãi.

Hàm strtok\_s khác với hàm strtok\_r của POSIX ở chỗ bảo vệ chống lưu trữ bên ngoài chuỗi được mã hóa và bằng cách kiểm tra các giới hạn thời gian chạy. Tuy nhiên, trên các chương trình được viết chính xác, strtok\_s và strtok\_r hoạt động giống nhau.

Sử dụng strtok\_s với ví dụ hiện mang lại phản hồi chính xác, như vậy:

## PHẦN 6.2 STRING LITERALS

Chuỗi hằng (string literals) trong C đại diện cho các mảng ký tự có kết thúc bằng ký tự null và có thời gian sống tĩnh (static-duration). Vì có thời gian sống tĩnh, một chuỗi hằng hoặc một con trỏ trỏ tới cùng một mảng gốc có thể an toàn sử dụng trong nhiều trường hợp mà một con trỏ tới mảng tự động không thể. Ví dụ, việc trả về một chuỗi hằng từ một hàm có hành vi xác định rõ ràng:

|  |
| --- |
| const char \*get\_hello() {  return "Hello, World!"; /\* an toàn \*/  } |

Do lưu trữ tĩnh, các chuỗi hằng không thể bị sửa đổi, và mọi cố gắng thay đổi các phần tử của mảng tương ứng của chuỗi hằng sẽ dẫn đến hành vi không xác định (undefined behavior). Thông thường, một chương trình cố gắng sửa đổi mảng tương ứng của chuỗi hằng sẽ gây ra lỗi hoặc không hoạt động đúng.

|  |
| --- |
| char \*foo = "hello";  foo[0] = 'y'; /\* Hành vi không xác định - KHÔNG TỐT! \*/ |

Để tránh việc sửa đổi không an toàn như vậy, khi một con trỏ trỏ tới một chuỗi hằng hoặc có thể trỏ tới nó, chúng ta nên khai báo con trỏ đó là **const** để tránh vô tình gây ra hành vi không xác định.

|  |
| --- |
| const char \*foo = "hello";  /\* TỐT: không thể sửa đổi chuỗi mà foo trỏ tới \*/ |

Tuy nhiên, một con trỏ trỏ vào hoặc trỏ vào mảng gốc của một chuỗi hằng không có tính đặc biệt về bản chất. Chúng ta có thể tự do sửa đổi giá trị của con trỏ để trỏ tới một địa chỉ khác.

|  |
| --- |
| char \*foo = "hello";  foo = "World!"; /\* OK - chúng ta chỉ đổi địa chỉ mà foo trỏ tới \*/ |

Mặt khác, việc khởi tạo một mảng ký tự bằng cách sử dụng cú pháp giống như chuỗi hằng không làm cho mảng đã khởi tạo có các đặc tính của chuỗi hằng. Khởi tạo chỉ đơn giản xác định độ dài và nội dung ban đầu của mảng. Các phần tử trong mảng này vẫn có thể được sửa đổi nếu chúng không được khai báo là hằng số.

|  |
| --- |
| char foo[] = "hello";  foo[0] = 'y'; /\* OK! \*/ |

## PHẦN 6.3 TÍNH ĐỘ DÀI CHUỖI: STRLEN()

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main(int *argc*, char \*\**argv*)  {  /\* Exit if no second argument is found. \*/  if (*argc* != 2)  {  puts("Argument missing.");  return EXIT\_FAILURE;  }  *size\_t* len = strlen(*argv*[1]);  printf("The length of the second argument is %zu.\\n", len);  return EXIT\_SUCCESS;  } |

Chương trình trên tính độ dài của **đối số thứ hai** mà nó nhận được và lưu kết quả vào biến **len**. Sau đó, nó in ra độ dài đó lên màn hình. Ví dụ, khi chạy với các tham số program\_name "Hello, world!", chương trình sẽ in ra **The length of the second argument is 13**. bởi vì chuỗi Hello, world! có độ dài là 13 ký tự.

Hàm **strlen** đếm tất cả các byte từ đầu chuỗi cho đến ký tự kết thúc **NULL ('\0')**. Vì vậy, nó chỉ có thể sử dụng khi đảm bảo chuỗi kết thúc bằng NULL.

Tuy nhiên, hãy lưu ý rằng nếu chuỗi chứa các ký tự Unicode, hàm **strlen** sẽ không cho bạn biết có bao nhiêu ký tự trong chuỗi (vì một số ký tự có thể dài hơn một byte). Trong trường hợp này, bạn phải tự đếm các ký tự (đơn vị mã) của chuỗi. Ví dụ dưới đây giải thích điều này:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main(void) {  char asciiString[50] = "Hello world!";  char utf8String[50] = "Γειά σου Κόσμε!"; /\* "Hello World!" in Greek \*/  printf("asciiString co %zu byte trong mang\n", sizeof(asciiString));  printf("utf8String co %zu byte trong mang\n", sizeof(utf8String));  printf("\"%s\" co %zu byte\n", asciiString, strlen(asciiString));  printf("\"%s\" co %zu byte\n", utf8String, strlen(utf8String));  return 0;  } |

**Kết quả**

|  |
| --- |
| asciiString co 50 byte trong mang  utf8String co 50 byte trong mang  "Hello world!" co 12 byte  "╬ô╬╡╬╣╬¼ ╧â╬┐╧à ╬Ü╧î╧â╬╝╬╡!" co 27 byte |

## PHẦN 6.4 GIỚI THIỆU CƠ BẢN VỀ CHUỖI

Trong ngôn ngữ lập trình C, một chuỗi (string) là một chuỗi các ký tự kết thúc bằng ký tự null (‘\0’).

Chúng ta có thể tạo các chuỗi sử dụng chuỗi ký tự (**string literals**), là các chuỗi ký tự được bao quanh bởi dấu ngoặc kép; ví dụ, **"hello world".** Chuỗi ký tự được tự động kết thúc bằng ký tự **null.**

Chúng ta có thể tạo các chuỗi bằng nhiều phương pháp khác nhau. Ví dụ, chúng ta có thể khai báo một con trỏ kiểu char và khởi tạo nó để trỏ đến ký tự đầu tiên của một chuỗi:

|  |
| --- |
| char \*string = "hello world"; |

Khi khởi tạo một con trỏ char bằng một chuỗi hằng như trên, chuỗi thực tế thường được cấp phát trong bộ nhớ chỉ đọc (read-only data); string là một con trỏ trỏ đến phần tử đầu tiên của mảng, tức là ký tự '**h**'.

Vì chuỗi hằng này được cấp phát trong bộ nhớ chỉ đọc, nó không thể thay đổi. Bất kỳ thay đổi nào của nó sẽ dẫn đến hành vi không xác định, do đó tốt hơn hãy thêm từ khóa const để có một thông báo lỗi tại thời gian biên dịch như sau:

|  |
| --- |
| char const \*string = "hello world"; |

Nó có tác dụng tương tự như sau:

|  |
| --- |
| char const string\_arr[] = "hello world"; |

Để tạo một chuỗi có thể sửa đổi, bạn có thể khai báo một mảng ký tự và khởi tạo nội dung của nó bằng chuỗi ký tự, như sau:

|  |
| --- |
| char modifiable\_string[] = "hello world"; |

Điều này tương đương với:

|  |
| --- |
| char modifiable\_string[] = {'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'}; |

Vì phiên bản thứ hai sử dụng cách khởi tạo bằng dấu ngoặc nhọn, chuỗi không tự động được kết thúc bằng ký tự null trừ khi ký tự **'\0'** được bao gồm rõ ràng trong mảng ký tự, thường là phần tử cuối cùng của mảng.

**Lưu ý:**

* Từ "không thể sửa đổi" (non-modifiable) chỉ ám chỉ rằng các ký tự trong chuỗi hằng không thể thay đổi, nhưng hãy nhớ rằng con trỏ string có thể được sửa đổi (có thể trỏ tới một vị trí khác hoặc tăng hoặc giảm giá trị).
* Cả hai chuỗi đều có hiệu quả tương tự trong việc không cho phép thay đổi các ký tự của chúng. Lưu ý rằng string là một con trỏ kiểu char và nó là một l-value có thể sửa đổi, nó có thể được tăng giá trị hoặc trỏ đến một vị trí khác trong khi mảng string\_arr là một l-value không thể sửa đổi, nó không thể thay đổi.

## PHẦN 6.5 SAO CHÉP CHUỖI

### 6.5.1 Gán con trỏ không sao chép chuỗi

Trong C, việc sử dụng toán tử = (gán) không sao chép chuỗi. Chuỗi trong C được biểu diễn dưới dạng mảng các ký tự kết thúc bằng ký tự null, vì vậy việc sử dụng toán tử = chỉ lưu địa chỉ (con trỏ) của chuỗi đó.

|  |
| --- |
| #include <stdio.h>  int main(void) {  int a = 10, b;  char c[] = "abc", \*d;  b = a; /\* Integer is copied \*/  a = 20; /\* Modifying a leaves b unchanged - b is a 'deep copy' of a \*/  printf("%d %d\n", a, b); /\* "20 10" will be printed \*/  d = c;  /\* Only copies the address of the string -  there is still only one string stored in memory \*/    c[1] = 'x';  /\* Modifies the original string - d[1] = 'x' will do exactly the same thing \*/  printf("%s %s\n", c, d); /\* "axc axc" will be printed \*/  return 0;  } |

Đoạn mã nguồn trên đã biên dịch thành công vì chúng ta sử dụng con trỏ char \*d thay vì mảng char d[3]. Khi sử dụng mảng char d[3], chương trình sẽ gặp lỗi biên dịch vì trong C, bạn không thể gán trực tiếp cho một mảng.

|  |
| --- |
| #include <stdio.h>  int main(void) {  char a[] = "abc";  char b[8];  b = a; /\* compile error \*/  printf("%s\n", b);  return 0;  } |

### 6.5.2 Sao chép chuỗi dùng các hàm tiêu chuẩn

#### 6.5.2.1 strcpy()

Để sao chép chuỗi, ta có thể sử dụng hàm strcpy() trong thư viện string.h. Trước khi sao chép, ta cần phải cấp phát đủ không gian cho chuỗi đích.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {  char a[] = "abc";  char b[8];  strcpy(b, a); /\* think "b special equals a" \*/  printf("%s\n", b); /\* "abc" will be printed \*/  return 0;  } |

#### 6.5.2.2 snprintf()

Để tránh tràn bộ đệm, snprintf() có thể được sử dụng. Đây không phải là giải pháp tốt nhất về mặt hiệu năng vì nó phải phân tích cú pháp chuỗi mẫu, nhưng đây là chức năng an toàn giới hạn bộ đệm duy nhất để sao chép các chuỗi có sẵn trong thư viện chuẩn, có thể được sử dụng mà không cần thực hiện thêm bất kỳ bước nào.

Hàm snprintf() hữu ích khi ta muốn sao chép một chuỗi vào một bộ đệm cố định với kích thước xác định mà không muốn gây ra lỗi tràn bộ đệm (buffer overflow). Thay vì sao chép toàn bộ nội dung của chuỗi như hàm strcpy(), hàm snprintf() cho phép bạn chỉ định kích thước tối đa cho bộ đệm đích.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {  char a[] = "012345678901234567890";  char b[8];  #if 0  strcpy(b, a); /\* causes buffer overrun (undefined behavior), so do not execute this here! \*/  #endif  snprintf(b, sizeof(b), "%s", a); /\* does not cause buffer overrun \*/  printf("%s\n", b); /\* "0123456" will be printed \*/  return 0;  } |

#### 6.5.2.3 strncat()

Một lựa chọn thứ hai, với hiệu suất tốt hơn, là sử dụng hàm strncat() (phiên bản kiểm tra tràn bộ đệm của strcat()) - nó nhận thêm một đối số thứ ba để xác định số byte tối đa cần sao chép.

|  |
| --- |
| char dest[32];  dest[0] = '\0'; // Đảm bảo dest là chuỗi rỗng ban đầu  strncat(*dest*, *source*, sizeof(dest) - 1);  // Sao chép tối đa (sizeof(dest) - 1) phần tử từ source vào dest  // Sau đó, thêm ký tự null '\0' vào cuối chuỗi dest |

Lưu ý rằng trong công thức trên, chúng ta sử dụng sizeof(dest) - 1. Điều này quan trọng vì hàm strncat() luôn thêm một ký tự null ('\0') vào cuối chuỗi, nhưng không tính ký tự này trong kích thước của chuỗi (điều này có thể gây nhầm lẫn và tràn bộ đệm).

Một ví dụ khác là khi ghép chuỗi sau một chuỗi không rỗng:

|  |
| --- |
| char dst[24] = "Clownfish: ";  char src[] = "Marvin and Nemo";  *size\_t* len = strlen(dst);  strncat(*dst*, *src*, sizeof(dst) - len - 1);  printf("%zu: [%s]\n", strlen(*dst*), *dst*); |

Kết quả khi in ra màn hình sẽ là:

|  |
| --- |
| 23: [Clownfish: Marvin and N] |

Lưu ý rằng kích thước được chỉ định là độ dài còn lại của mảng đích (không tính ký tự null cuối cùng). Điều này có thể gây ra vấn đề ghi đè lớn hơn. Để xác định đúng độ dài cần sao chép, bạn nên chỉ định địa chỉ của ký tự null cuối cùng trong nội dung hiện có của chuỗi đích, giúp strncat() không phải quét lại chuỗi đích trước khi bắt đầu sao chép.

|  |
| --- |
| strcpy(*dst*, "Clownfish: ");  assert(len < sizeof(dst) - 1);  strncat(dst + len, src, sizeof(dst) - len - 1);  printf("%zu: [%s]\n", strlen(*dst*), *dst*); |

Điều này tạo ra cùng kết quả như trước, nhưng strncat() không cần quét lại nội dung hiện có của dst trước khi bắt đầu sao chép.

#### 6.5.2.4 strncpy()

Hàm strncpy() là lựa chọn cuối cùng. Mặc dù có thể nghĩ rằng nó nên đứng đầu, nhưng nó lại là một hàm gây nhầm lẫn với hai vấn đề chính:

* Nếu sao chép bằng strncpy() vượt quá giới hạn của bộ đệm, một ký tự null cuối cùng sẽ không được sao chép vào đích.
* strncpy() luôn điền đầy đủ nội dung của đích, bằng các ký tự null nếu cần thiết.

Hiện tại, việc triển khai kỳ quặc này có lịch sử và ban đầu được dùng để xử lý tên tập tin UNIX.)

Cách duy nhất để sử dụng strncpy() đúng đắn là tự đảm bảo kết thúc chuỗi bằng ký tự null:

|  |
| --- |
| strncpy(*b*, *a*, sizeof(b)); /\* tham số thứ ba là kích thước của bộ đệm đích \*/  b[sizeof(b)/sizeof(\*b) - 1] = '\0'; /\* kết thúc chuỗi \*/  printf("%s\n", b); /\* "0123456" sẽ được in ra \*/ |

Tuy nhiên, thậm chí khi bạn có một bộ đệm lớn, việc sử dụng strncpy() trở nên rất không hiệu quả do cần phải điền ký tự null thừa.

## LẶP QUA CÁC KÍ TỰ TRONG MỘT CHUỖI

Nếu chúng ta biết độ dài của chuỗi, chúng ta có thể sử dụng vòng lặp for để lặp lại các ký tự của chuỗi:

|  |
| --- |
| char \* string = "hello world"; /\* This 11 chars long, excluding the 0-terminator. \*/  *size\_t* i = 0;  for (; i < 11; i++) {  printf("%c\n", string[i]); /\* Print each character of the string. \*/  } |

Ngoài ra, chúng ta có thể sử dụng hàm tiêu chuẩn strlen() để lấy độ dài của một chuỗi nếu chúng ta không biết chuỗi đó là gì:

|  |
| --- |
| size\_t length = strlen(string);  *size\_t* i = 0;  for (; i < length; i++) {  printf("%c\n", string[i]); /\* Print each character of the string. \*/  } |

Cuối cùng, chúng ta có thể tận dụng lợi thế của thực tế là các chuỗi trong C được đảm bảo là kết thúc null (điều mà chúng ta đã làm khi chuyển nó sang strlen() trong ví dụ trước ;-)). Chúng tôi có thể lặp lại mảng bất kể kích thước của nó và dừng lặp lại khi chúng tôi đạt đến ký tự null:

|  |
| --- |
| size\_t i = 0;  while (string[i] != '\0') { /\* Stop looping when we reach the null-character. \*/  printf("%c\n", string[i]); /\* Print each character of the string. \*/  i++;  } |

## TẠO MẢNG CHUỖI

Một mảng chuỗi trong C có thể có ý nghĩa như sau:

* Một mảng có các phần tử là các con trỏ char \*.
* Một mảng có các phần tử là các mảng ký tự.

Chúng ta có thể tạo một mảng các con trỏ ký tự như sau:

|  |
| --- |
| char \* string\_array[] = {  "foo",  "bar",  "baz"  }; |

Hãy nhớ rằng khi gán các chuỗi chữ cho con trỏ **char \***, các chuỗi đó thực sự được lưu trữ trong bộ nhớ chỉ đọc (**read-only**). Tuy nhiên, mảng string\_array được lưu trữ trong bộ nhớ có thể đọc và viết (**read/write**). Điều này có nghĩa là chúng ta có thể chỉnh sửa các con trỏ trong mảng, nhưng không thể chỉnh sửa các chuỗi mà chúng trỏ tới.

Trong C, tham số argv của hàm main (mảng các tham số dòng lệnh được truyền khi chạy chương trình) là một mảng các con trỏ char \*: char \*argv[].

Chúng ta cũng có thể tạo mảng các mảng ký tự. Vì chuỗi là mảng ký tự, một mảng chuỗi đơn giản là một mảng có các phần tử là các mảng ký tự:

|  |
| --- |
| char modifiable\_string\_array\_literals[][4] = {  "foo",  "bar",  "baz"  }; |

Điều này tương đương với:

|  |
| --- |
| char modifiable\_string\_array[][4] = {  {'f', 'o', 'o', '\0'},  {'b', 'a', 'r', '\0'},  {'b', 'a', 'z', '\0'}  }; |

Lưu ý rằng chúng ta chỉ định 4 là kích thước của chiều thứ hai của mảng; mỗi chuỗi trong mảng của chúng ta thực sự có kích thước 4 byte vì chúng ta phải bao gồm ký tự kết thúc chuỗi **(null-terminating character).**

## CHUYỂN ĐỔI CHUỖI THÀNH SỐ: ATOI(), ATOF() (CẨN THẬN KHÔNG NÊN SỬ DỤNG)

Cảnh báo: Các hàm atoi, atol, atoll và atof inherently là không an toàn, bởi vì: Nếu giá trị của kết quả không thể biểu diễn, hành vi của chúng không xác định. (7.20.1p1)

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int *argc*, char\*\* *argv*)  {  int val;  if (*argc* < 2)  {  printf("Usage: %s <integer>\n", *argv*[0]);  return 0;  }  val = atoi(*argv*[1]);  printf("String value = %s, Int value = %d\n", *argv*[1], val);  return 0;  } |

Khi chuỗi cần chuyển đổi là một số nguyên hợp lệ và nằm trong phạm vi của kiểu dữ liệu int, thì hàm atoi() hoạt động đúng:

|  |
| --- |
| $ ./atoi 100  String value = 100, Int value = 100  $ ./atoi 200  String value = 200, Int value = 200 |

Tuy nhiên, nếu chuỗi bắt đầu bằng một số, tiếp theo là ký tự khác, thì chỉ số đầu tiên của chuỗi đó sẽ được chuyển đổi:

|  |
| --- |
| $ ./atoi 0x200  0  $ ./atoi 0123x300  123 |

Nếu chuỗi không phải là một số hợp lệ hoặc không nằm trong phạm vi kiểu dữ liệu int, thì hành vi của hàm atoi() sẽ là không xác định, có thể gây ra lỗi hoặc dẫn đến các hậu quả không mong muốn. Ví dụ:

|  |
| --- |
| $ ./atoi hello  Formatting the hard disk... |

Do sự mơ hồ ở trên và hành vi không xác định này, họ các chức năng atoi không bao giờ nên được sử dụng.

Để chuyển đổi thành long int, hãy sử dụng strtol() thay vì atol().

Để chuyển đổi thành double, hãy sử dụng strtod() thay vì atof().

Để chuyển đổi thành long long int, hãy sử dụng strtoll() thay vì atoll().

## ĐỌC GHI DỮ LIỆU DẠNG CHUỖI

Ghi dữ liệu đã định dạng vào chuỗi

|  |
| --- |
| int sprintf ( char \* *str*, const char \* *format*, ... ); |

sử dụng hàm **sprintf** để ghi dữ liệu float vào chuỗi.

|  |
| --- |
| #include <stdio.h>  int main ()  {  char buffer [50];  double PI = 3.1415926;  sprintf (buffer, "PI = %.7f", PI);  printf ("%s\n",buffer);  return 0;  } |

Kết quả

|  |
| --- |
| PI = 3.1415926 |

Đọc dữ liệu được định dạng từ chuỗi

|  |
| --- |
| int sscanf ( const char \* *s*, const char \* *format*, ...); |

sử dụng hàm **sscanf** để phân tích dữ liệu được định dạng.

|  |
| --- |
| #include <stdio.h>  int main ()  {  char sentence []="date : 06-06-2012";  char str [50];  int year;  int month;  int day;  sscanf (sentence,"%s : %2d-%2d-%4d", str, &day, &month, &year);  printf ("%s -> %02d-%02d-%4d\n",str, day, month, year);  return 0;  } |

Kết quả:

|  |
| --- |
| date -> 06-06-2012 |

## TÌM VỊ TRÍ XUẤT HIỆN ĐẦU TIÊN/ CUỐI CÙNG CỦA MỘT KÍ TỰ CỤ THỂ: STRCHR(), STRRCHR()

Các hàm **strchr** và **strrchr** tìm một ký tự trong một chuỗi, ký tự đó nằm trong một mảng ký tự kết thúc bằng NULL. **strchr** trả về một con trỏ tới lần xuất hiện đầu tiên và **strrchr** tới lần xuất hiện cuối cùng.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  int main(int *argc*, char \**argv*[]) // Thêm khai báo biến argc và argv ở đây  {  char toSearchFor = 'A';  /\* Exit if no second argument is found. \*/  if (*argc* != 2)  {  printf("Argument missing.\n");  return EXIT\_FAILURE;  }  {  char \*firstOcc = strchr(*argv*[1], toSearchFor);  if (firstOcc != NULL)  {  printf("First position of %c in %s is %td.\n",  toSearchFor, *argv*[1], firstOcc - *argv*[1]);  /\* A pointer difference's result is a signed integer and uses the length modifier 't'. \*/  }  else  {  printf("%c is not in %s.\n", toSearchFor, *argv*[1]);  }  }  {  char \*lastOcc = strrchr(*argv*[1], toSearchFor);  if (lastOcc != NULL)  {  printf("Last position of %c in %s is %td.\n",  toSearchFor, *argv*[1], lastOcc - *argv*[1]);  }  }  return EXIT\_SUCCESS;  } |

Kết quả đầu ra (sau khi đã tạo một pos có tên thực thi):

|  |
| --- |
| $ ./pos AAAAAAA  First position of A in AAAAAAA is 0.  Last position of A in AAAAAAA is 6.  $ ./pos BAbbbbbAccccAAAAzzz  First position of A in BAbbbbbAccccAAAAzzz is 1.  Last position of A in BAbbbbbAccccAAAAzzz is 15.  $ ./pos qwerty  A is not in qwerty. |

Một cách sử dụng phổ biến cho strrchr là trích xuất tên tệp từ một đường dẫn. Ví dụ: để giải nén myfile.txt từ C:\Users\eak\myfile.txt:

|  |
| --- |
| char \*getFileName(const char \**path*)  {  char \*pend;  if ((pend = strrchr(*path*, '\')) != NULL)  return pend + 1;  return NULL;  } |

## SAO CHÉP VÀ NỐI CHUỖI: STRCPY(), STRCAT()

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {  /\* Khai báo một mảng có đủ độ dài để chứa các ký tự và ký tự null kết thúc chuỗi ('\0'). \*/  char mystring[10];  /\* Sao chép chuỗi "foo" vào `mystring` cho đến khi gặp ký tự null. \*/  strcpy(mystring, "foo");  printf("%s\n", mystring);  /\* Tại thời điểm này, chúng ta đã sử dụng 4 ký tự trong `mystring`, bao gồm 3 ký tự của "foo" và ký tự null kết thúc. \*/  /\* Nối chuỗi "bar" vào `mystring`. \*/  strcat(mystring, "bar");  printf("%s\n", mystring);  /\* Bây giờ chúng ta đã sử dụng 7 ký tự trong `mystring`: "foo" dùng 3 ký tự, "bar" dùng 3 ký tự và ký tự null kết thúc. \*/  /\* Sao chép chuỗi "bar" vào `mystring`, ghi đè nội dung cũ. \*/  strcpy(mystring, "bar");  printf("%s\n", mystring);  return 0;  } |

Kết quả:

|  |
| --- |
| foo  foobar  bar |

Khi bạn nối vào, sao chép từ hoặc nối từ một chuỗi hiện có, hãy đảm bảo rằng chuỗi này đã được kết thúc bằng ký tự null trước khi thực hiện thao tác. Trường hợp của bạn ví dụ, chuỗi từ hằng số (string literals) như "foo" đã được biên dịch sẽ luôn được kết thúc bằng ký tự null bởi trình biên dịch, vì vậy bạn có thể yên tâm sử dụng chúng trong các hàm xử lý chuỗi.

## SO SÁNH: STRCMP(), STRNCMP(), STRCASECMP(), STRNCASECMP()

Các hàm strcase\*-không phải là Tiêu chuẩn C, mà là phần mở rộng POSIX.

strcmp(): Hàm này so sánh hai chuỗi ký tự kết thúc bằng null (null-terminated) theo thứ tự từ điển. Hàm trả về một giá trị số nguyên:

* Nếu chuỗi thứ nhất xuất hiện trước chuỗi thứ hai trong thứ tự từ điển, hàm trả về một giá trị âm.
* Nếu hai chuỗi bằng nhau, hàm trả về giá trị 0.
* Nếu chuỗi thứ nhất xuất hiện sau chuỗi thứ hai trong thứ tự từ điển, hàm trả về một giá trị dương.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  void compare(char const \**lhs*, char const \**rhs*)  {  int result = strcmp(*lhs*, *rhs*); // compute comparison once  if (result < 0) {  printf("%s comes before %s\n", *lhs*, *rhs*);  } else if (result == 0) {  printf("%s equals %s\n", *lhs*, *rhs*);  } else { // last case: result > 0  printf("%s comes after %s\n", *lhs*, *rhs*);  }  }  int main(void)  {  compare("BBB", "BBB");  compare("BBB", "CCCCC");  compare("BBB", "AAAAAA");  return 0;  } |

Kết quả:

|  |
| --- |
| BBB equals BBB  BBB comes before CCCCC  BBB comes after AAAAAA |

Giống như hàm **strcmp**, hàm **strcasecmp** cũng so sánh các đối số của nó theo từ điển sau khi dịch từng ký tự sang chữ thường tương ứng của nó:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  void compare(char const \**lhs*, char const \**rhs*)  {  int result = strcasecmp(*lhs*, *rhs*); // compute case-insensitive comparison once  if (result < 0) {  printf("%s comes before %s\n", *lhs*, *rhs*);  } else if (result == 0) {  printf("%s equals %s\n", *lhs*, *rhs*);  } else { // last case: result > 0  printf("%s comes after %s\n", *lhs*, *rhs*);  }  }  int main(void)  {  compare("BBB", "bBB");  compare("BBB", "ccCCC");  compare("BBB", "aaaaaa");  return 0;  } |

Kết quả:

|  |
| --- |
| BBB equals bBB  BBB comes before ccCCC  BBB comes after aaaaaa |

**strncmp** và **strncasecmp** so sánh tối đa n ký tự. Hàm này nhận thêm một tham số thứ ba là số lượng ký tự để so sánh:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  void compare(char const \**lhs*, char const \**rhs*, int *n*)  {  int result = strncmp(*lhs*, *rhs*, *n*); // compute comparison once  if (result < 0) {  printf("%s comes before %s\n", *lhs*, *rhs*);  } else if (result == 0) {  printf("%s equals %s\n", *lhs*, *rhs*);  } else { // last case: result > 0  printf("%s comes after %s\n", *lhs*, *rhs*);  }  }  int main(void)  {  compare("BBB", "Bb", 1);  compare("BBB", "Bb", 2);  compare("BBB", "Bb", 3);  return 0;  } |

Kết quả:

|  |
| --- |
| BBB equals Bb  BBB comes before Bb  BBB comes before Bb |

## CHUYỂN ĐỔI CHUỖI THÀNH SỐ MỘT CÁCH AN TOÀN: HÀM STRTOX

Kể từ C99, thư viện C có một tập hợp các hàm chuyển đổi an toàn diễn giải một chuỗi thành một số. Tên của chúng có dạng strtoX, trong đó X là một trong số l, ul, d, v.v. để xác định loại mục tiêu của chuyển đổi.

strtol: Chuyển đổi chuỗi sang số nguyên dạng long (32-bit hoặc 64-bit tùy thuộc vào hệ điều hành và kiến trúc).

strtoul: Chuyển đổi chuỗi sang số nguyên không dấu dạng unsigned long.

strtoll: Chuyển đổi chuỗi sang số nguyên dạng long long (64-bit).

strtoull: Chuyển đổi chuỗi sang số nguyên không dấu dạng unsigned long long.

strtod: Chuyển đổi chuỗi sang số thực dạng double.

|  |
| --- |
| double strtod(char const\* *p*, char\*\* *endptr*);  long double strtold(char const\* *p*, char\*\* *endptr*); |

Chúng cung cấp khả năng kiểm tra xem chuyển đổi có bị tràn hay thiếu không:

|  |
| --- |
| double ret = strtod(argv[1], 0); /\* attempt conversion \*/  /\* check the conversion result. \*/  if ((ret == HUGE\_VAL || ret == -HUGE\_VAL) && errno == ERANGE)  return; /\* numeric overflow in in string \*/  else if (ret == HUGE\_VAL && errno == ERANGE)  return; /\* numeric underflow in in string \*/  /\* At this point we know that everything went fine so ret may be used \*/ |

Nếu chuỗi không chứa số nào, hàm **strtod** trên sẽ trả về giá trị 0.0.

Tuy nhiên, nếu không muốn nhận giá trị mặc định này, ta có thể sử dụng tham số bổ sung **endptr**. Tham số này là một con trỏ tới con trỏ, và nó sẽ được trỏ tới cuối số đã được phát hiện trong chuỗi. Nếu tham số này được đặt thành 0 hoặc NULL, nó sẽ được bỏ qua.

Sử dụng tham số **endptr**, ta có thể kiểm tra xem quá trình chuyển đổi có thành công hay không và xác định vị trí kết thúc của số trong chuỗi:

|  |
| --- |
| char \*check = 0;  double ret = strtod(argv[1], &*check*); /\* thử chuyển đổi \*/  if (argv[1] == check)  return; /\* Không phát hiện số nào trong chuỗi \*/  else if ((ret == HUGE\_VAL || ret == -HUGE\_VAL) && errno == ERANGE)  return; /\* Tràn số khi chuyển đổi \*/  else if (ret == HUGE\_VAL && errno == ERANGE)  return; /\* Số quá nhỏ khi chuyển đổi \*/  /\* Ở đây, ta biết rằng mọi thứ đã diễn ra suôn sẻ, giá trị ret có thể sử dụng \*/ |

Có các hàm tương tự để chuyển đổi sang các loại số nguyên rộng hơn:

|  |
| --- |
| long strtol(char const\* *p*, char\*\* *endptr*, int *nbase*);  long long strtoll(char const\* *p*, char\*\* *endptr*, int *nbase*);  unsigned long strtoul(char const\* *p*, char\*\* *endptr*, int *nbase*);  unsigned long long strtoull(char const\* *p*, char\*\* *endptr*, int *nbase*); |

Các hàm này có tham số thứ ba nbase giữ cơ sở số trong đó số được viết.

|  |
| --- |
| long a = strtol("101", 0, 2 ); /\* a = 5L \*/  long b = strtol("101", 0, 8 ); /\* b = 65L \*/  long c = strtol("101", 0, 10); /\* c = 101L \*/  long d = strtol("101", 0, 16); /\* d = 257L \*/  long e = strtol("101", 0, 0 ); /\* e = 101L \*/  long f = strtol("0101", 0, 0 ); /\* f = 65L \*/  long g = strtol("0x101", 0, 0 ); /\* g = 257L \*/ |

Giá trị 0 cho nbase có nghĩa là chuỗi sẽ được hiểu như các số trong chương trình C: tiền tố 0x tương ứng với số thập lục phân, tiền tố 0 tương ứng với số bát phân, và tất cả các số khác được hiểu là số thập phân.

Do đó, cách thiết thực nhất để diễn giải một đối số dòng lệnh dưới dạng số sẽ là

|  |
| --- |
| int main(int *argc*, char\* *argv*[] {  if (*argc* < 1)  return EXIT\_FAILURE; /\* No number given. \*/  /\* use strtoull because size\_t may be wide \*/  *size\_t* mySize = strtoull(*argv*[1], 0, 0);  /\* then check conversion results. \*/  ...  return EXIT\_SUCCESS;  } |

Điều này cho phép chương trình được gọi với một tham số dòng lệnh ở hệ thống bát phân, thập phân hoặc thập lục phân.

## STRSPN VÀ STRCSPN

Cho trước một chuỗi, strspn tính toán độ dài của chuỗi con ban đầu (span) chỉ bao gồm một danh sách ký tự cụ thể. strcspn cũng tương tự, ngoại trừ nó tính toán độ dài của chuỗi con ban đầu bao gồm bất kỳ ký tự nào ngoại trừ những ký tự được liệt kê:

|  |
| --- |
| /\*  Provided a string of "tokens" delimited by "separators", print the tokens along  with the token separators that get skipped.  \*/  #include <stdio.h>  #include <string.h>  int main(void)  {  const char sepchars[] = ",.;!?";  char foo[] = ";ball call,.fall gall hall!?.,";  char \*s;  int n;  for (s = foo; \*s != 0; /\*empty\*/) {  /\* Get the number of token separator characters. \*/  n = (int)strspn(s, sepchars);  if (n > 0)  printf("skipping separators: << %.\*s >> (length=%d)\n", n, s, n);  /\* Actually skip the separators now. \*/  s += n;  /\* Get the number of token (non-separator) characters. \*/  n = (int)strcspn(s, sepchars);  if (n > 0)  printf("token found: << %.\*s >> (length=%d)\n", n, s, n);  /\* Skip the token now. \*/  s += n;  }  printf("== token list exhausted ==\n");  return 0;  } |

Các hàm tương tự sử dụng chuỗi ký tự rộng là wcsspn và wcscspn; chúng được sử dụng theo cùng một cách.

# CHỮ SỐ, KÝ TỰ VÀ CHUỖI

## FLOATING POINT LITERALS( CHỮ SỐ DẤU CHẤM ĐỘNG)

Chữ dấu chấm động được sử dụng để biểu diễn các số thực có dấu. Các hậu tố sau đây có thể được sử dụng để chỉ định loại chữ:

|  |  |  |
| --- | --- | --- |
| Hậu tố | Kiểu dữ liệu | Ví dụ |
| Không có | double | 3.1415926 -3E6 |
| f, F | float | 3.1415926f 2.1E-6F |
| l, L | long double | 3.1415926L 1E126L |

Để sử dụng các hậu tố này, hằng số phải là một hằng số số thực dấu chấm động. Ví dụ, 3f là không hợp lệ, vì 3 là một hằng số số nguyên, trong khi 3.f hoặc 3.0f là đúng. Đối với kiểu long double, khuyến nghị luôn sử dụng chữ hoa L để cải thiện tính đọc hiểu.

## STRING LITERALS( CHUỖI KÍ TỰ CHỮ)

Chuỗi ký tự chữ được sử dụng để xác định các mảng ký tự. Chúng là các chuỗi ký tự được bao quanh bởi dấu ngoặc kép (ví dụ: "abcd") và có kiểu dữ liệu là char\*.

Tiền tố L làm cho chữ trở thành một mảng ký tự rộng, thuộc loại wchar\_t\*. Ví dụ: L"abcd".

Kể từ phiên bản C11, có các tiền tố mã hóa khác, tương tự như tiền tố L:

|  |  |  |
| --- | --- | --- |
| Tiền tố | Loại cơ sở | Mã hóa |
| Không có | char | tùy thuộc vào nền tảng |
| L | wchar\_t | tùy thuộc vào nền tảng |
| u8 | char | UTF-8 |
| u | char16\_t | Thường là UTF-16 |
| U | char32\_t | Thường là UTF-32 |

Đối với hai tiền tố cuối cùng, ta có thể kiểm tra với các macro kiểm tra tính năng xem liệu mã hóa có phải là mã hóa UTF tương ứng hay không.

## CHARACTER LITERALS(KÝ TỰ)

Hằng ký tự là một loại đặc biệt của hằng số số nguyên được sử dụng để biểu diễn một ký tự duy nhất. Chúng được bao quanh bởi dấu ngoặc đơn, ví dụ: 'a' và có kiểu dữ liệu là int. Giá trị của hằng số là một giá trị số nguyên tương ứng với bộ ký tự của máy tính. Chúng không cho phép tiền tố.

Tiền tố L trước một hằng ký tự biến nó thành một ký tự rộng của kiểu wchar\_t. Tương tự, từ phiên bản C11, các tiền tố u và U biến chúng thành ký tự rộng của kiểu char16\_t và char32\_t.

Khi muốn biểu diễn một số ký tự đặc biệt như ký tự không in, ta sử dụng các chuỗi escape. Các chuỗi escape sử dụng một chuỗi ký tự được dịch thành một ký tự khác. Tất cả các chuỗi escape đều bao gồm hai hoặc nhiều ký tự, ký tự đầu tiên trong chuỗi là dấu gạch chéo . Các ký tự ngay sau dấu gạch chéo xác định ký tự mà chuỗi escape được hiểu là gì.

Các chuỗi escape được sử dụng để biểu diễn các ký tự như sau:

|  |  |
| --- | --- |
| \b | Dấu phẩy ngược |
| \f | Lùi trang |
| \n | Xuống dòng |
| \r | Cú pháp |
| \t | Tab ngang |
| \v | Tab dọc |
| \\ | Dấu gạch chéo ngược |
| \’ | Dấu nháy đơn |
| \” | Dấu nháy kép |
| \? | Dấu chấm hỏi |
| \nnn | Giá trị bát phân |
| \xnn | Giá trị thập lục phân |
| \a | Ký tự thông báo (còi) |
| \unnnn | Tên ký tự toàn cầu (universal character name) có 4 chữ số thập lục phân |
| \Unnnnnnnn | Tên ký tự toàn cầu có 8 chữ số thập lục phân |

Một tên ký tự toàn cầu là một mã Unicode. Một tên ký tự toàn cầu có thể ánh xạ thành nhiều ký tự. Các chữ số n được hiểu là các chữ số thập lục phân. Tùy thuộc vào mã hóa UTF đang sử dụng, một chuỗi tên ký tự toàn cầu có thể cho ra một mã Unicode bao gồm nhiều ký tự, thay vì chỉ một ký tự duy nhất.

Khi sử dụng chuỗi escape xuống dòng trong I/O chế độ văn bản, nó sẽ được chuyển đổi thành byte xuống dòng cụ thể của hệ điều hành.

Chuỗi escape dấu chấm hỏi được sử dụng để tránh trigraphs. Ví dụ, ??/ sẽ được biên dịch thành trigraph đại diện cho ký tự gạch chéo '', nhưng sử dụng ??/ sẽ cho ra chuỗi "??/". Có thể có một, hai hoặc ba số octal n trong chuỗi escape giá trị bát phân.

## INTEGER LITERALS(SỐ NGUYÊN)

Hằng số nguyên được sử dụng để cung cấp các giá trị nguyên. Có ba cơ số số học được hỗ trợ, được chỉ định bằng tiền tố:

|  |  |  |
| --- | --- | --- |
| Cơ số | Tiền tố | Ví dụ |
| Thập phân | Không có | 5 |
| Bát phân | 0 | 0345 |
| Thập lục phân | 0x hoặc 0X | 0x12AB, 0X12AB, 0x12ab, 0x12Ab |

Lưu ý rằng viết không bao gồm bất kỳ dấu hiệu nào, do đó hằng số nguyên luôn luôn dương. Một số như -1 được xem như là biểu thức có một hằng số nguyên (1) được đảo dấu bằng dấu -.

Kiểu dữ liệu của hằng số nguyên thập phân là kiểu dữ liệu đầu tiên có thể chứa giá trị từ int và long. Kể từ C99, kiểu long long cũng được hỗ trợ cho các hằng số rất lớn.

Kiểu dữ liệu của hằng số nguyên bát phân hoặc thập lục phân là kiểu dữ liệu đầu tiên có thể chứa giá trị từ int, unsigned, long và unsigned long. Kể từ C99, kiểu long long và unsigned long long cũng được hỗ trợ cho các hằng số rất lớn.

Sử dụng các hậu tố khác nhau, kiểu mặc định của hằng số có thể được thay đổi.

|  |  |
| --- | --- |
| Hậu tố | Giải thích |
| L, l | long int |
| LL, ll( kể tử C99) | long long int |
| U,u | unsigned |

Các hậu tố U và L/LL có thể được kết hợp trong bất kỳ thứ tự và chữ hoa nào. Lỗi nếu có hậu tố bị trùng lặp (ví dụ cung cấp hai hậu tố U) ngay cả khi chúng có chữ hoa khác nhau.

# COMPOUND LITERALS

## ĐỊNH NGHĨA/ KHỞI TẠO COMPOUND LITERALS

A compound literal là một đối tượng vô danh được tạo ra trong phạm vi nơi nó được định nghĩa. Khái niệm này được giới thiệu lần đầu trong tiêu chuẩn C99. Dưới đây là một ví dụ về A compound literal:

**Ví dụ từ tiêu chuẩn C, C11-§6.5.2.5/9:**

|  |
| --- |
| int \*p = (int [2]){ 2, 4 }; |

Biến p được khởi tạo với địa chỉ của phần tử đầu tiên của một mảng vô danh gồm hai số nguyên.

Hằng số hợp thành là một lvalue. Tuổi thọ của đối tượng vô danh được xác định bởi thời gian sống của nó, có thể là tĩnh (nếu hằng số hợp thành xuất hiện trong phạm vi tệp tin) hoặc tự động (nếu hằng số hợp thành xuất hiện trong phạm vi khối), và trong trường hợp sau, đối tượng sẽ kết thúc khi điều khiển thoát khỏi khối bao quanh nó.

|  |
| --- |
| void f(void)  {  int \*p;  /\*...\*/  p = (int [2]){ \*p };  /\*...\*/  } |

Biến p được gán địa chỉ của phần tử đầu tiên của một mảng gồm hai số nguyên, phần tử đầu tiên có giá trị trước đó được trỏ bởi p và phần tử thứ hai có giá trị là 0. Trong trường hợp này, biến p vẫn còn hiệu lực cho đến khi kết thúc khối.

**Compound literal với bộ chỉ mục**

|  |
| --- |
| struct *point* {  unsigned x;  unsigned y;  };  extern void drawline(struct *point*, struct *point*);  // Sử dụng nơi đó  drawline((struct point){.x=1, .y=1}, (struct point){.x=3, .y=4}); |

Một hàm giả định drawline nhận hai đối số có kiểu struct point. Đối số đầu tiên có giá trị tọa độ x == 1 và y == 1, trong khi đối số thứ hai có x == 3 và y == 4.

**Compound literal không xác định chiều dài của mảng**

|  |
| --- |
| int \*p = (int []){ 1, 2, 3}; |

Trong trường hợp này, kích thước của mảng không được xác định trước và sẽ được xác định bởi độ dài của danh sách khởi tạo.

**Compound literal có độ dài của danh sách khởi tạo nhỏ hơn kích thước của mảng đã chỉ định**

|  |
| --- |
| int \*p = (int [10]){1, 2, 3}; |

Các phần tử còn lại của Compound literal sẽ được khởi tạo mặc định là 0.

**Compound literal chỉ đọc**

Lưu ý rằng Compound literal là một lvalue và do đó các phần tử của nó có thể thay đổi được. Compound literal chỉ đọc có thể được chỉ định bằng cách sử dụng từ khóa const như sau:

|  |
| --- |
| (const int[]){1, 2} |

Compound literal chứa các biểu thức tùy ý

Trong một hàm, Compound literal, giống như bất kỳ khởi tạo nào từ tiêu chuẩn C99 trở đi, có thể chứa các biểu thức tùy ý.

|  |
| --- |
| void foo()  {  int \*p;  int i = 2; j = 5;  /\*...\*/  p = (int [2]){ i+j, i\*j };  /\*...\*/  } |

Trong trường hợp này, biến p được gán địa chỉ của phần tử đầu tiên của một mảng gồm hai số nguyên, với phần tử đầu tiên có giá trị là i+j và phần tử thứ hai có giá trị là i\*j.

# BIT- FIENDS

# MẢNG

Mảng là các loại dữ liệu kiểu dữ liệu dẫn xuất (derived data type) là kiểu dữ liệu (data type) có nguồn gốc từ các kiểu dữ liệu nguyên thủy (primitive data type) hoặc kiểu dữ liệu xây dựng sẵn (built-in data type).

Trong C có bốn loại, cụ thể là: Hàm (function), mảng (array), con trỏ (pointer), tham chiếu (reference), đại diện cho một tập hợp có thứ tự các giá trị ("phần tử") của một loại dữ liệu khác.

Hầu hết các mảng trong C có một số lượng cố định các phần tử của bất kỳ loại nào và biểu diễn của chúng lưu trữ các phần tử liền kề trong bộ nhớ mà không có khoảng trống hoặc ghi chú. C cho phép mảng đa chiều, trong đó các phần tử là các mảng khác, và cũng cho phép mảng của con trỏ.

C hỗ trợ các mảng được cấp phát động có kích thước được xác định tại thời điểm chạy. C99 và các phiên bản sau đó hỗ trợ các mảng có độ dài biến đổi hay còn gọi là VLA (Variable Length Arrays).

## Khai báo và khởi tạo mảng

Cú pháp chung để khai báo mảng một chiều là

|  |
| --- |
| type arrName[size]; |

Ở đây, **type** có thể là bất kỳ kiểu dữ liệu cơ bản nào hoặc kiểu dữ liệu do người dùng định nghĩa, ví dụ như các structures, **arrName** là một định danh do người dùng định nghĩa và **size** là một hằng số nguyên.

**VD 1**: Khai báo một mảng gồm 10 biến kiểu int.

|  |
| --- |
| int array[10]; |

Hiện tại nó chứa các giá trị không xác định. Để đảm bảo nó chứa các giá trị 0 khi khai báo, bạn có thể làm như sau:

|  |
| --- |
| int array[10] = {0}; |

Mảng cũng có thể có bộ khởi tạo, ví dụ này khai báo một mảng gồm 10 số nguyên, trong đó 3 số nguyên đầu tiên sẽ chứa các giá trị 1, 2, 3, các giá trị còn lại sẽ là 0:

|  |
| --- |
| int array[10] = {1, 2, 3}; |

Trong phương pháp khởi tạo trên, giá trị đầu tiên trong danh sách sẽ được gán cho phần tử đầu tiên của mảng, giá trị thứ hai sẽ được gán cho thành viên thứ hai của mảng và cứ tiếp tục như vậy. Nếu kích thước của danh sách nhỏ hơn kích thước của mảng, thì như trong ví dụ trên, các thành viên còn lại của mảng sẽ được khởi tạo thành giá trị 0.

Với cách khởi tạo danh sách có chỉ định (ISO C99), khởi tạo rõ ràng cho các thành viên của mảng là có thể. Ví dụ:

|  |
| --- |
| int array[5] = {[2] = 5, [1] = 2, [4] = 9}; /\* mảng là {0, 2, 5, 0, 9} \*/ |

Trong hầu hết các trường hợp, trình biên dịch có thể suy ra độ dài của mảng cho bạn, điều này có thể được đạt bằng cách để trống các dấu ngoặc vuông:

|  |
| --- |
| int array[] = {1, 2, 3}; /\* một mảng gồm 3 số nguyên \*/  int array[] = {[3] = 8, [0] = 9}; /\* kích thước là 4 \*/ |

Khai báo một mảng có độ dài bằng 0 là không được phép.

Phiên bản C99 trở lên hỗ trợ Mảng Động (Variable Length Arrays - VLA). Tuy nhiên, từ phiên bản C11 trở đi, VLA đã trở thành tùy chọn. VLA tương đương với mảng thông thường, nhưng có một điểm khác biệt quan trọng: Độ dài không cần phải biết tại thời điểm biên dịch. VLA có thời gian lưu trữ tự động. Chỉ có con trỏ tới VLA có thể có thời gian lưu trữ tĩnh.

|  |
| --- |
| *size\_t* m = tinh\_do\_dai(); /\* tính độ dài mảng tại thời điểm chạy \*/  int vla[m]; /\* tạo mảng với độ dài tính toán được \*/ |

**Quan trọng:**

VLA có thể nguy hiểm. Nếu mảng vla trong ví dụ trên yêu cầu nhiều không gian trên ngăn xếp hơn khả dụng, thì ngăn xếp sẽ tràn. Việc sử dụng VLA thường không được khuyến khích trong các hướng dẫn về phong cách và sách giáo trình và bài tập.

## Lặp qua mảng hiệu quả và thứ tự ưu tiên hàng( row major order)

Mảng trong ngôn ngữ lập trình C có thể được xem như một vùng nhớ liên tục. Cụ thể hơn, chiều cuối cùng của mảng là phần liên tục trong bộ nhớ. Chúng ta gọi điều này là "thứ tự ưu tiên hàng" (row-major order). Điều quan trọng là hiểu được và thực tế rằng khi xảy ra "cache fault" (lỗi bộ nhớ đệm), bộ nhớ đệm sẽ tải một "dòng bộ nhớ đệm" hoàn chỉnh vào bộ nhớ đệm khi truy cập dữ liệu chưa được lưu vào bộ nhớ đệm, nhằm ngăn lỗi bộ nhớ đệm sau này. Vì vậy, khi truy cập vào một mảng có kích thước 10000x10000 và bắt đầu từ **array[0][0],** có thể sẽ tải **array[0][1]** vào bộ nhớ đệm, để tránh lỗi bộ nhớ đệm trong trường hợp truy cập tiếp theo. Tuy nhiên, nếu truy cập ngay sau đó là **array[1][0]**, thì điều này sẽ tạo ra một lỗi bộ nhớ đệm thứ hai. Điều này xảy ra vì **array[1][0]** cách **array[0][0]** một khoảng **sizeof(type)\*10000 byte**, và do đó, chắc chắn không nằm trên cùng một dòng bộ nhớ đệm. Vì vậy, khi lặp qua mảng một cách không hiệu quả như sau:

|  |
| --- |
| #define ARRLEN 10000  int array[ARRLEN][ARRLEN];  *size\_t* i, j;  for (i = 0; i < ARRLEN; ++i)  {  for(j = 0; j < ARRLEN; ++j)  {  array[j][i] = 0;  }  } |

Cách hiệu quả hơn là:

|  |
| --- |
| #define ARRLEN 10000  int array[ARRLEN][ARRLEN];  *size\_t* i, j;  for (i = 0; i < ARRLEN; ++i)  {  for(j = 0; j < ARRLEN; ++j)  {  array[i][j] = 0;  }  } |

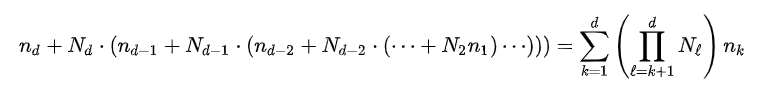
Cũng giống như trên, khi làm việc với mảng một chiều có nhiều chỉ số (ví dụ: 2 chiều với chỉ số i và j), bạn nên lặp qua mảng như sau:

|  |
| --- |
| #define DIM\_X 10  #define DIM\_Y 20  int array[DIM\_X\*DIM\_Y];  *size\_t* i, j;  for (i = 0; i < DIM\_X; ++i)  {  for(j = 0; j < DIM\_Y; ++j)  {  array[i\*DIM\_Y+j] = 0;  }  } |

Và với 3 chiều và chỉ số i, j, k:

|  |
| --- |
| #define DIM\_X 10  #define DIM\_Y 20  #define DIM\_Z 30  int array[DIM\_X\*DIM\_Y\*DIM\_Z];  *size\_t* i, j, k;  for (i = 0; i < DIM\_X; ++i)  {  for(j = 0; j < DIM\_Y; ++j)  {  for (k = 0; k < DIM\_Z; ++k)  {  array[i\*DIM\_Y\*DIM\_Z+j\*DIM\_Z+k] = 0;  }  }  } |

Hoặc một cách tổng quát hơn, khi chúng ta có một mảng với N1 x N2 x ... x Nd phần tử, d chiều và chỉ số được ký hiệu là n1, n2,..., nd, sự dịch chuyển được tính như sau:



Hình ảnh / công thức lấy từ: https://en.wikipedia.org/wiki/Row-major\_order

## Độ dài mảng

Độ dài của mảng có giá trị cố định và được xác định khi khai báo mảng. Tuy nhiên, ta có thể tính độ dài của mảng trong một số trường hợp để làm mã nguồn linh hoạt hơn, đặc biệt khi độ dài được xác định tự động từ bộ khởi tạo:

|  |
| --- |
| int array[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  /\* Kích thước của “array tính bằng byte \*/  *size\_t* size = sizeof(array);  /\* Số phần tử trong “array” \*/  *size\_t* length = sizeof(array) / sizeof(array[0]); |

Tuy nhiên trong hầu hết các ngữ cảnh mà mảng xuất hiện trong biểu thức, nó sẽ tự động chuyển đổi thành một con trỏ trỏ tới phần tử đầu tiên của mảng (gọi là "decay to pointer"). Trường hợp mảng là toán hạng của toán tử sizeof là một trong số ít các ngoại lệ. Con trỏ thu được **không phải là một mảng** và không mang bất kỳ thông tin về độ dài của mảng gốc. Vì vậy, nếu độ dài này cần thiết khi sử dụng con trỏ, ví dụ như khi con trỏ được truyền vào một hàm, thì nó phải được truyền riêng biệt.

Ví dụ, giả sử chúng ta muốn viết một hàm để trả về phần tử cuối cùng của một mảng int. Tiếp tục từ trên, chúng ta có thể gọi nó như vậy:

|  |
| --- |
| /\* Mảng 'array' sẽ tự động chuyển đổi thành một con trỏ,  \*vì vậy kích thước cần được truyền riêng biệt \*/  int last = get\_last(array, length); |

Và đây là cách triển khai hàm **get\_last**:

|  |
| --- |
| int get\_last(int *input*[], *size\_t* *length*) {  return input[length - 1];  } |

Lưu ý đặc biệt là khai báo của tham số **input** giống như khai báo một mảng, nhưng thực tế thì nó được xem như là một **con trỏ** tới kiểu **int**. Điều này tương đương với việc khai báo **int \*input**. Thậm chí nếu kích thước của mảng được chỉ định, thì điều này không ảnh hưởng đến kiểu của input. Điều này có thể được xem như một kỹ thuật nhớ dễ dàng.

Tuy nhiên, không nên cố gắng xác định kích thước của mảng từ con trỏ, như trong hàm **BAD\_get\_last**:

|  |
| --- |
| int BAD\_get\_last(int input[]) {  /\* SAI VÌ KHÔNG THỂ TÍNH KÍCH THƯỚC CỦA MẢNG MÀ input TRỎ TỚI: \*/  size\_t length = sizeof(input) / sizeof(input[0]);  return input[length - 1]; /\* Lỗi - không phải kết quả mong muốn \*/  } |

Trên thực tế, lỗi cụ thể đó phổ biến đến mức một số trình biên dịch nhận ra và cảnh báo về nó. clang, chẳng hạn, sẽ phát ra cảnh báo sau:

Sử dụng toán tử sizeof trên con trỏ không trả về kích thước của mảng mà nó trỏ tới, mà trả về kích thước của con trỏ. Điều này dẫn đến kết quả sai khi cố gắng truy cập vào phần tử cuối cùng của mảng. Do đó, để sử dụng đúng kích thước của mảng trong hàm, bạn phải truyền kích thước riêng biệt vào hàm như trong hàm get\_last.

|  |
| --- |
| warning: sizeof on array function parameter will return size of 'int \*' instead of 'int []' [-  Wsizeof-array-argument]  int length = sizeof(input) / sizeof(input[0]);  ^  note: declared here  int BAD\_get\_last(int input[])  ^ |

## Truyền mảng đa chiều vào hàm

Mảng đa chiều tuân theo cùng các quy tắc như mảng một chiều khi truyền chúng vào hàm. Tuy nhiên, sự kết hợp giữa decay-to-pointer, ưu tiên toán tử và hai cách khai báo mảng đa chiều (mảng của mảng và mảng của con trỏ) có thể khiến khai báo của các hàm như vậy trở nên không rõ ràng. Ví dụ sau đây chỉ ra các cách đúng để truyền mảng đa chiều vào hàm:

|  |
| --- |
| #include <assert.h>  #include <stdlib.h>  /\* Khi truyền một mảng đa chiều (tức là một mảng của các mảng) vào một hàm,  nó sẽ bị chuyển đổi thành con trỏ đến phần tử đầu tiên như thường lệ.  Nhưng chỉ có cấp cao nhất sẽ bị chuyển đổi, do đó cái được truyền vào là con trỏ đến một mảng với một kích thước cố định  (trong trường hợp này là 4). \*/  void f(int *x*[][4]) {  assert(sizeof(\**x*) == sizeof(int) \* 4);  }  /\* Nguyên mẫu này tương đương với f(int x[][4]).  Dấu ngoặc đơn xung quanh \*x là bắt buộc vì [index]  có mức độ ưu tiên cao hơn so với \*expr, do đó int \*x[4] thông thường sẽ tương đương với int  \*(x[4]), tức là một mảng gồm 4 con trỏ tới int.  Nhưng nếu nó được khai báo là một tham số của hàm,  nó sẽ bị chuyển đổi thành con trỏ và trở thành int \*\*x,  điều này không tương thích với x[2][4]. \*/  void g(int (\**x*)[4]) {  assert(sizeof(\**x*) == sizeof(int) \* 4);  }  /\* Một mảng của con trỏ có thể được truyền vào hàm này,  vì nó sẽ bị chuyển đổi thành con trỏ đến con trỏ (int \*\*),  nhưng một mảng của các mảng không thể. \*/  void h(int \*\**x*) {  assert(sizeof(\**x*) == sizeof(int\*));  }  int main(void) {  int foo[2][4];  f(foo);  g(foo);  /\* Ở đây, chúng ta đang tạo động một mảng của các con trỏ.  Lưu ý rằng kích thước của mỗi chiều không phải là một phần của kiểu dữ liệu,  vì vậy hệ thống kiểu chỉ coi nó như một con trỏ đến con trỏ (int \*\*),  không phải con trỏ đến mảng hay mảng của các mảng. \*/  int \*\*bar = malloc(sizeof(\*bar) \* 2);  assert(bar);  for (*size\_t* i = 0; i < 2; i++) {  bar[i] = malloc(sizeof(\*bar[i]) \* 4);  assert(bar[i]);  }  h(bar);  for (*size\_t* i = 0; i < 2; i++) {  free(bar[i]);  }  free(bar);  } |

**Xem thêm**

Truyền mảng vào hàm

## Mảng đa chiều

Ngôn ngữ lập trình C cho phép sử dụng mảng đa chiều. Dưới đây là cú pháp chung của khai báo mảng đa chiều:

|  |
| --- |
| type name[size1][size2]...[sizeN]; |

**Ví dụ 1:** Khai báo sau tạo ra một mảng số nguyên ba chiều (5 x 10 x 4):

|  |
| --- |
| int arr[5][10][4]; |

### Mảng hai chiều

Dạng đơn giản nhất của mảng đa chiều là mảng hai chiều. Mảng hai chiều thực chất là một danh sách của các mảng một chiều. Để khai báo một mảng hai chiều số nguyên với kích thước m x n, chúng ta có thể viết như sau:

|  |
| --- |
| type arrayName[m][n]; |

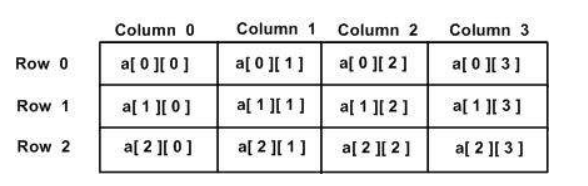
Trong đó:

type : là bất kỳ kiểu dữ liệu hợp lệ trong C (int, float, v.v...) và

arrayName có thể là bất kỳ định danh hợp lệ trong C.

Một mảng hai chiều có thể được hình dung như một bảng với m hàng và n cột. Lưu ý rằng thứ tự cần phải đúng trong C. Mảng int a[4][3] không giống với mảng int a[3][4]. Số hàng được đặt trước vì C là một ngôn ngữ ưu tiên hàng đầu.

Một mảng hai chiều a, chứa ba hàng và bốn cột có thể được biểu diễn như sau:



Do đó, mỗi phần tử trong mảng arrayName được xác định bởi tên phần tử có dạng arrayName[m][n]. Trong đó **arrayName** là tên của mảng, **m** là chỉ số hàng và **n** là chỉ số cột. Nhớ rằng các hàng và cột được đánh số từ 0. Điều này rất giống với cách đánh chỉ số trong toán học cho ma trận 2D.

#### Khởi tạo mảng 2 chiều

Mảng đa chiều có thể được khởi tạo bằng cách chỉ định các giá trị trong ngoặc đơn cho mỗi hàng.

**Ví dụ 2**: Định nghĩa một mảng có tên là array với 3 hàng, mỗi hàng có 4 cột.

|  |
| --- |
| int array[3][4] = {  {0, 1, 2, 3}, /\* giá trị khởi tạo cho hàng có chỉ số 0 \*/  {4, 5, 6, 7}, /\* giá trị khởi tạo cho hàng có chỉ số 1 \*/  {8, 9, 10, 11} /\* giá trị khởi tạo cho hàng có chỉ số 2 \*/  }; |

Dấu ngoặc nhọn lồng nhau, biểu thị hàng được chọn, là tùy chọn. Khởi tạo sau đây tương đương với ví dụ trước:

|  |
| --- |
| int array[3][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}}; |

Trong khi việc tạo mảng với ngoặc nhọn lồng nhau là tùy chọn, nó được khuyến khích mạnh mẽ vì nó dễ đọc và rõ ràng hơn.

#### Truy cập phần tử trong mảng hai chiều

Một phần tử trong mảng hai chiều được truy cập bằng cách sử dụng các chỉ mục con, tức là chỉ số hàng và chỉ số cột của mảng.

**Ví dụ 3:** Lấy giá trị của phần tử có vị trí hang thứ 3 và cột thứ 4.

|  |
| --- |
| int value = a[2][3]; |

Chúng ta sử dụng một vòng lặp lồng nhau để xử lý một mảng hai chiều:

|  |
| --- |
| #include <stdio.h>  int main () {    /\* một mảng với 5 hàng và 2 cột \*/  int array[5][2] = { {0, 0}, {1, 2}, {2, 4}, {3, 6}, {4, 8} };  int i, j;  /\* in giá trị của từng phần tử trong mảng \*/  for (i = 0; i < 5; i++) {  for (j = 0; j < 2; j++) {  printf("a[%d][%d] = %d\n", i, j, a[i][j]);  }  }  return 0;  } |

Khi đoạn mã trên được biên dịch và thực thi, nó sẽ tạo ra kết quả như sau:

|  |
| --- |
| a[0][0] = 0  a[0][1] = 0  a[1][0] = 1  a[1][1] = 2  a[2][0] = 2  a[2][1] = 4  a[3][0] = 3  a[3][1] = 6  a[4][0] = 4  a[4][1] = 8 |

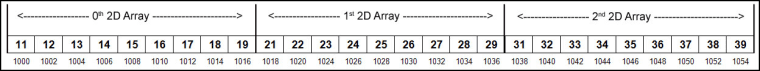
### Mảng ba chiều

Mảng 3D về cơ bản là một mảng của các mảng: đó là một tập hợp các mảng 2D và mảng 2D là một mảng gồm các mảng 1D.

A diagram of a number

Description automatically generated

#### Bộ nhớ mảng ba chiều



#### Khởi tạo một mảng ba chiều

|  |
| --- |
| double cprogram[3][2][4] = {  {{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},  {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},  {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}  }; |

Chúng ta có thể có mảng với bất kỳ số chiều nào, tuy nhiên thường thì hầu hết các mảng mà ta tạo ra sẽ có một hoặc hai chiều.

## Define mảng và truy cập vào phần tử mảng

|  |
| --- |
| #include <stdio.h>  #define ARRLEN (10)  int main(void) {  int n[ARRLEN]; // n là một mảng gồm 10 số nguyên  int i, j; // Sử dụng i, j để chỉ định vị trí bộ nhớ (địa chỉ) trong mảng  // Khởi tạo giá trị cho các phần tử trong mảng n  for (i = 0; i < ARRLEN; i++) {  n[i] = i + 100; // Đặt giá trị cho phần tử tại vị trí i là i + 100  }  // In giá trị của từng phần tử trong mảng n  for (j = 0; j < ARRLEN; j++) {  printf("Phan tu [%zu] = %d\n", j, n[j]);  }  return 0;  } |

**Kết quả**

|  |
| --- |
| Phan tu [0] = 100  Phan tu [1] = 101  Phan tu [2] = 102  Phan tu [3] = 103  Phan tu [4] = 104  Phan tu [5] = 105  Phan tu [6] = 106  Phan tu [7] = 107  Phan tu [8] = 108  Phan tu [9] = 109 |

## Xóa nội dung của mảng( Thiết lập giá trị bằng 0)

Đôi khi cần thiết phải đặt mảng về giá trị 0 sau khi đã khởi tạo nó.

|  |
| --- |
| #include <stdlib.h> /\* for EXIT\_SUCCESS \*/  #define ARRLEN (10)  int main(void) {  /\* Đã cấp phát nhưng chưa khởi tạo, vì không được xác định là static hoặc global. \*/  int array[ARRLEN];  *size\_t* i;  for (i = 0; i < ARRLEN; ++i) {  array[i] = 0;  }  return EXIT\_SUCCESS;  } |

Một phương pháp rút gọn chung cho vòng lặp trên là sử dụng **memset()** từ **<string.h>.** Truyền **array** như được hiển thị bên dưới sẽ làm cho nó rơi vào một con trỏ đến phần tử đầu tiên của nó.

|  |
| --- |
| memset(*array*, 0, ARRLEN \* sizeof(int)); /\* Sử dụng kiểu dữ liệu cụ thể (int ở đây). \*/ |

Hoặc

|  |
| --- |
| memset(*array*, 0, ARRLEN \* sizeof \**array*); /\* Sử dụng kích thước của kiểu dữ liệu mà con trỏ đang trỏ tới. \*/ |

Vì trong ví dụ này array là một mảng và không chỉ là một con trỏ đến phần tử đầu tiên của mảng (xem Phần 10.3: Độ dài mảng để hiểu tại sao điều này quan trọng), một tùy chọn thứ ba để thiết lập giá trị 0 cho mảng là:

|  |
| --- |
| memset(*array*, 0, sizeof *array*); /\* Sử dụng kích thước của mảng chính nó. \*/ |

## Thiết lập giá trị trong mảng

Truy cập các giá trị trong mảng thường được thực hiện thông qua dấu ngoặc vuông:

|  |
| --- |
| int value;  int array[10];  /\* Thiết lập giá trị của phần tử thứ năm có giá trị là 5: \*/  array[4] = 5;  /\* Cách viết trên tương đương với: \*/  \*(array + 4) = 5;  /\* Đọc giá trị của phần tử thứ năm: \*/  value = array[4]; |

Do tính chất của các toán hạng của toán tử + có thể hoán đổi vị trí (--> luật giao hoán), việc sau đây cũng tương đương:

|  |
| --- |
| \*(array + 4) = 5;  \*(4 + array) = 5; |

Tương tự, các câu lệnh sau cũng tương đương:

|  |
| --- |
| array[4] = 5;  4[array] = 5; /\* Kỳ lạ nhưng hợp lệ trong C ... \*/ |

Và các câu lệnh dưới đây cũng tương tự nhau:

|  |
| --- |
| val = array[4];  val = 4[array]; /\* Kỳ lạ nhưng hợp lệ trong C ... \*/ |

Trong C, không thực hiện bất kỳ kiểm tra ranh giới nào, việc truy cập nội dung bên ngoài mảng đã được khai báo sẽ không xác định (truy cập vào bộ nhớ vượt quá phần được cấp phát):

|  |
| --- |
| int value;  int array[10];  array[4] = 5; /\* ổn \*/  value = array[4]; /\* ổn \*/  array[19] = 20; /\* hành vi không xác định \*/  value = array[15]; /\* hành vi không xác định \*/ |

## Cấp phát và khởi tạo giá trị ban đầu cho một mảng với kích thước do người dùng xác định

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main (void) {  int \*pdata;  *size\_t* n;  printf("Nhap kich thuoc cua mang: ");  fflush(stdout); /\* Đảm bảo in ra thông báo trên stdout được đệm. \*/  if (1 != scanf("%zu", &n)) /\* Nếu không hỗ trợ "zu" (trên Windows?), sử dụng "lu". \*/  {  fprintf("scanf() khong doc duoc gia tri hop le\n");  exit(EXIT\_FAILURE);  }  pdata = calloc(n, sizeof \*pdata);  if (NULL == pdata) {  perror("calloc() that bai"); /\* In ra thông báo lỗi. \*/  exit(EXIT\_FAILURE);  }  free(pdata); /\* Giải phóng bộ nhớ. \*/  return EXIT\_SUCCESS;  } |

Chương trình này cố gắng đọc vào một giá trị số nguyên không dấu từ đầu vào tiêu chuẩn, sau đó cấp phát một khối bộ nhớ cho một mảng có n phần tử kiểu int bằng cách gọi hàm calloc(). Bộ nhớ này được khởi tạo giá trị 0 cho tất cả các phần tử. Trong trường hợp thành công, bộ nhớ được giải phóng bằng cách gọi hàm free().

## Duyệt qua mảng bằng con trỏ

|  |
| --- |
| #include <stdio.h>  #define SIZE (10)  int main() {  *size\_t* i = 0;  int \*p = NULL;  int array[SIZE];    /\* Thiết lập giá trị cho mảng a là i\*i \*/  for(i = 0; i < SIZE; ++i)  {  array[i] = i \* i;  }    /\* Đọc các giá trị bằng cách sử dụng con trỏ \*/  for(p = array; p < array + SIZE; ++p)  {  printf("%d\n", \*p);  }  return 0;  } |

Trong đoạn mã này, trong việc khởi tạo biến p trong điều kiện vòng lặp for đầu tiên, mảng array sẽ rơi vào trạng thái con trỏ tới phần tử đầu tiên của nó, như làm vậy ở hầu hết các nơi mà biến mảng được sử dụng.

Sau đó, ++p thực hiện tính toán con trỏ trên con trỏ **p** và duyệt qua từng phần tử của mảng và trỏ đến chúng bằng cách giải tham chiếu với **\*p**.

# LINKED LISTS

## DOUBLY LINKED LIST (DANH SÁCH LIÊN KẾT KÉP)

Dưới đây là một ví dụ về mã nguồn hiển thị cách chèn các nút vào danh sách liên kết kép, cách đảo ngược danh sách và cách in danh sách theo thứ tự ngược.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  /\* Dữ liệu này không phải lúc nào cũng được lưu trữ trong một cấu trúc,  \*nhưng đôi khi nó được dùng để dễ sử dụng \*/  struct *Node* {  /\* Đôi khi một khóa cũng được lưu trữ và sử dụng trong các chức năng \*/  int data;  struct *Node*\* next;  struct *Node*\* previous;  };  void insert\_at\_beginning(struct *Node* \*\**pheadNode*, int *value*);  void insert\_at\_end(struct *Node* \*\**pheadNode*, int *value*);  void print\_list(struct *Node* \**headNode*);  void print\_list\_backwards(struct *Node* \**headNode*);  void free\_list(struct *Node* \**headNode*);  int main(void) {  /\* Đôi khi trong danh sách liên kết kép, nút cuối cùng cũng được lưu trữ \*/  struct *Node* \*head = NULL;    printf("Chen mot nut vao dau danh sach.\n");  insert\_at\_beginning(&head, 5);  print\_list(head);    printf("Chen mot nut vao dau, sau do in danh sach theo thu tu nguoc\n");  insert\_at\_beginning(&head, 10);  print\_list\_backwards(head);    printf("Chen mot nut vao cuoi, sau do in danh sach theo thu tu xuoi.\n");  insert\_at\_end(&head, 15);  print\_list(head);    free\_list(head);  return 0;  }  void print\_list\_backwards(struct *Node* \**headNode*) {  if (NULL == *headNode*) {  return;  }  /\*  Lặp qua danh sách và khi chúng ta đi đến cuối,  lặp ngược lại để in ra các mục theo thứ tự ngược lại  (điều này được thực hiện với con trỏ tới nút trước đó).  Điều này thậm chí có thể được thực hiện dễ dàng hơn  nếu một con trỏ tới nút cuối cùng được lưu trữ.  \*/  struct *Node* \*i = *headNode*;  while (i->next != NULL) {  i = i->next;/\* Di chuyển đến cuối danh sách \*/  }  while (i != NULL) {  printf("Gia tri: %d\n", i->data);  i = i->previous;  }  }  void print\_list(struct *Node* \**headNode*) {  /\* Duyệt qua danh sách và in ra thành viên dữ liệu của mỗi nút \*/  struct *Node* \*i;  for (i = *headNode*; i != NULL; i = i->next) {  printf("Gia tri: %d\n", i->data);  }  }  void insert\_at\_beginning(struct *Node* \*\**pheadNode*, int *value*) {  struct *Node* \*currentNode;  if (NULL == *pheadNode*) {  return;  }  /\*  Điều này được thực hiện tương tự như cách chúng ta chèn một nút vào đầu danh sách liên kết đơn,  thay vào đó chúng ta cũng đặt thành viên trước đó của cấu trúc  \*/  currentNode = malloc(sizeof \*currentNode);  currentNode->next = NULL;  currentNode->previous = NULL;  currentNode->data = *value*;  if (\**pheadNode* == NULL) { /\* Danh sách rỗng \*/  \**pheadNode* = currentNode;  return;  }  currentNode->next = \**pheadNode*;  (\**pheadNode*)->previous = currentNode;  \**pheadNode* = currentNode;  }  void insert\_at\_end(struct *Node* \*\**pheadNode*, int *value*) {  struct *Node* \*currentNode;  if (NULL == *pheadNode*) {  return;  }  /\*  Điều này một lần nữa có thể được thực hiện dễ dàng bằng cách có thể có phần tử trước đó.  Nó cũng sẽ hữu ích hơn nếu có một con trỏ tới nút cuối cùng, thường được sử dụng.  \*/  currentNode = malloc(sizeof \*currentNode);  struct *Node* \*i = \**pheadNode*;  currentNode->data = *value*;  currentNode->next = NULL;  currentNode->previous = NULL;  if (\**pheadNode* == NULL) {  \**pheadNode* = currentNode;  return;  }  while (i->next != NULL) { /\* Đi đến cuối danh sách \*/  i = i->next;  }  i->next = currentNode;  currentNode->previous = i;  }  void free\_list(struct *Node* \**node*) {  while (*node* != NULL) {  struct *Node* \*next = *node*->next;  free(*node*);  *node* = next;  }  } |

Lưu ý rằng đôi khi, việc lưu trữ một con trỏ tới nút cuối cùng là hữu ích (chỉ đơn giản là có thể chuyển thẳng đến cuối danh sách sẽ hiệu quả hơn là cần phải lặp lại từ đầu đến cuối):

|  |
| --- |
| struct *Node* \*lastNode = NULL; |

Trong trường hợp đó, cần cập nhật nó khi thay đổi danh sách.

Đôi khi, một khóa cũng được sử dụng để xác định các phần tử. Nó chỉ đơn giản là một thành viên của cấu trúc Node:

|  |
| --- |
| struct *Node* {  int data;  int key;  struct *Node*\* next;  struct *Node*\* previous;  }; |

Sau đó, khóa này được sử dụng khi bất kỳ tác vụ nào được thực hiện trên một phần tử cụ thể, chẳng hạn như xóa phần tử

## REVERSING A LINKED LIST (ĐẢO NGƯỢC DANH SÁCH LIÊN KẾT)

Bạn cũng có thể thực hiện nhiệm vụ này theo cách đệ quy, nhưng tôi đã chọn trong ví dụ này để sử dụng phương pháp lặp. Tác vụ này hữu ích nếu bạn đang chèn tất cả các nút của mình vào đầu danh sách được liên kết. Đây là một ví dụ:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define NUM\_ITEMS 10  struct *Node* {  int data;  struct *Node* \*next;  };  void chen\_nut(struct *Node* \*\**dauDanhSach*, int *giaTriNut*, int *viTri*);  void in\_danh\_sach(struct *Node* \**dauDanhSach*);  void dao\_nguoc\_danh\_sach(struct *Node* \*\**dauDanhSach*);  int main(void) {  int i;  struct *Node* \*dauDanhSach = NULL;    // Tạo danh sách ban đầu từ 1 đến NUM\_ITEMS  for(i = 1; i <= NUM\_ITEMS; i++) {  chen\_nut(&dauDanhSach, i, i);  }    // In ra danh sách ban đầu  printf("Danh sach ban dau:\n");  in\_danh\_sach(dauDanhSach);    // Đảo ngược danh sách  printf("Danh sach sau khi dao nguoc:\n");  dao\_nguoc\_danh\_sach(&dauDanhSach);  in\_danh\_sach(dauDanhSach);    return 0;  }  // Hàm chèn một nút mới vào danh sách liên kết đơn tại vị trí đã cho  void chen\_nut(struct *Node* \*\**dauDanhSach*, int *giaTriNut*, int *viTri*) {  int i;  struct *Node* \*nutHienTai = (struct *Node* \*)malloc(sizeof(struct *Node*));  struct *Node* \*nutTruocViTri = \**dauDanhSach*;  nutHienTai->data = *giaTriNut*;    // Nếu vị trí là 1, thì nút mới trở thành nút đầu danh sách  if(*viTri* == 1) {  nutHienTai->next = \**dauDanhSach*;  \**dauDanhSach* = nutHienTai;  return;  }    // Di chuyển đến nút trước vị trí cần chèn  for(i = 0; i < *viTri* - 2; i++) {  nutTruocViTri = nutTruocViTri->next;  }    nutHienTai->next = nutTruocViTri->next;  nutTruocViTri->next = nutHienTai;  }  // Hàm in ra các giá trị trong danh sách liên kết đơn  void in\_danh\_sach(struct *Node* \**dauDanhSach*) {  struct *Node* \*dieuHuong;  for(dieuHuong = *dauDanhSach*; dieuHuong != NULL; dieuHuong = dieuHuong->next) {  printf("Gia tri: %d\n", dieuHuong->data);  }  }  // Hàm đảo ngược danh sách liên kết đơn  void dao\_nguoc\_danh\_sach(struct *Node* \*\**dauDanhSach*) {  struct *Node* \*dieuHuong = \**dauDanhSach*;  struct *Node* \*nutTruoc = NULL;  struct *Node* \*nutSau = NULL;    while(dieuHuong != NULL) {  nutSau = dieuHuong->next;  dieuHuong->next = nutTruoc;  nutTruoc = dieuHuong;  dieuHuong = nutSau;  }    \**dauDanhSach* = nutTruoc;  } |

**Giải thích cho phương pháp REVERSING A LINKED LIST (ĐẢO NGƯỢC DANH SÁCH LIÊN KẾT)**

Chúng tôi bắt đầu nutTruoc là NULL, vì chúng tôi biết trong lần lặp đầu tiên của vòng lặp, nếu chúng tôi đang tìm kiếm nút trước nút đầu tiên, thì nó sẽ là NULL. Nút đầu tiên sẽ trở thành nút cuối cùng trong danh sách và biến tiếp theo đương nhiên phải là NULL.

Về cơ bản, khái niệm đảo ngược danh sách liên kết ở đây là chúng ta thực sự đảo ngược chính các liên kết. Thành viên tiếp theo của mỗi nút sẽ trở thành nút trước nó, giống như:

|  |
| --- |
| Head -> 1 -> 2 -> 3 -> 4 -> 5 |

Trong đó mỗi số đại diện cho một nút. Danh sách này sẽ trở thành:

|  |
| --- |
| 1 <- 2 <- 3 <- 4 <- 5 <- Head |

Cuối cùng, thay vào đó, phần đầu sẽ trỏ đến nút thứ 5 và mỗi nút sẽ trỏ đến nút trước nó.

Nút 1 phải trỏ đến NULL vì không có gì trước nó. Nút 2 phải trỏ đến nút 1, nút 3 phải trỏ đến nút 2, v.v.

Tuy nhiên, có một vấn đề nhỏ với phương pháp này. Nếu chúng ta ngắt liên kết đến nút tiếp theo và thay đổi nó thành nút trước đó, chúng ta sẽ không thể đi qua nút tiếp theo trong danh sách vì liên kết đến nó đã biến mất.

Giải pháp cho vấn đề này là chỉ cần lưu trữ phần tử tiếp theo trong một biến (nextNode) trước khi thay đổi liên kết.

## INSERTING A NODE AT THE NTH POSITION (CHÈN NÚT Ở VỊ TRÍ THỨ N)

Cho đến giờ, chúng ta đã xem xét việc chèn một nút vào đầu danh sách liên kết đơn. Tuy nhiên, hầu hết các trường hợp bạn cũng muốn có thể chèn các nút ở nơi khác. Đoạn mã được viết dưới đây cho thấy cách có thể viết hàm insert() để chèn các nút vào bất kỳ đâu trong danh sách được liên kết.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  struct *Node* {  int data;  struct *Node*\* next;  };  struct *Node*\* insert(struct *Node*\* *head*, int *value*, *size\_t* *position*);  void print\_list(struct *Node*\* *head*);  int main(int *argc*, char\* *argv*[]) {  struct *Node*\* head = NULL; /\* Khởi tạo danh sách rỗng \*/  /\* Chèn các node vào vị trí và giá trị tương ứng: \*/  head = insert(head, 1, 0);  head = insert(head, 100, 1);  head = insert(head, 21, 2);  head = insert(head, 2, 3);  head = insert(head, 5, 4);  head = insert(head, 42, 2);  print\_list(head);    return 0;  }  struct *Node*\* insert(struct *Node*\* *head*, int *value*, *size\_t* *position*) {  *size\_t* i = 0;  struct *Node*\* currentNode;  /\* Tạo node mới \*/  currentNode = malloc(sizeof \*currentNode);  /\* Kiểm tra thành công của malloc() ở đây! \*/    /\* Gán giá trị dữ liệu \*/  currentNode->data = *value*;  /\* Con trỏ trỏ tới trường 'next' mà chúng ta phải liên kết với node mới.  \* Bằng cách khởi tạo nó thành &head, chúng ta xử lý trường hợp chèn vào đầu danh sách. \*/  struct *Node*\*\* nextForPosition = &*head*;  /\* Lặp để lấy trường 'next' mà chúng ta đang tìm kiếm.  \* Lưu ý: Chèn vào cuối nếu vị trí lớn hơn số phần tử hiện tại. \*/  for (i = 0; i < *position* && \*nextForPosition != NULL; i++) {  /\* nextForPosition trỏ tới trường 'next' của node.  \* Vì vậy, \*nextForPosition là một con trỏ tới node tiếp theo.  \* Cập nhật nó thành một con trỏ tới trường 'next' của node tiếp theo. \*/  nextForPosition = &(\*nextForPosition)->next;  }  /\* Ở đây, chúng ta lấy liên kết tới node tiếp theo (node mà node mới của chúng ta nên trỏ tới)  \* bằng cách giải tham chiếu nextForPosition, con trỏ này trỏ tới trường 'next' của node  \* đang ở vị trí chúng ta muốn chèn node mới vào.  \* Chúng ta gán liên kết này cho giá trị next của node mới. \*/  currentNode->next = \*nextForPosition;  /\* Bây giờ, chúng ta muốn sửa liên kết của node trước vị trí của node mới:  \* nó sẽ được thay đổi thành con trỏ tới node mới của chúng ta. \*/  \*nextForPosition = currentNode;  return *head*;  }  void print\_list(struct *Node*\* *head*) {  /\* Duyệt qua danh sách các node và in ra giá trị dữ liệu trong mỗi node \*/  struct *Node*\* i = *head*;  while (i != NULL) {  printf("%d\n", i->data);  i = i->next;  }  } |

## INSERTING A NODE AT THE BEGINNING OF A SINGLY LINKED LIST (CHÈN NÚT VÀO ĐẦU DANH SÁCH LIÊN KẾT ĐƠN)

Đoạn mã dưới đây sẽ nhắc các số và tiếp tục thêm chúng vào đầu danh sách được liên kết.

|  |
| --- |
| /\* Chương trình này sẽ thể hiện cách chèn một node vào đầu danh sách liên kết \*/  #include <stdio.h>  #include <stdlib.h>  struct *Node* {  int data;  struct *Node*\* next;  };  void insert\_node(struct *Node*\*\* *head*, int *nodeValue*);  void print\_list(struct *Node*\* *head*);  int main(int *argc*, char\* *argv*[]) {  struct *Node*\* headNode;  headNode = NULL; /\* Khởi tạo con trỏ của node đầu tiên thành NULL. \*/  *size\_t* listSize, i;  do {  printf("Nhập số lượng số bạn muốn nhập:\n");  } while (1 != scanf("%zu", &listSize));  for (i = 0; i < listSize; i++) {  int numToAdd;  do {  printf("Nhập một số:\n");  } while (1 != scanf("%d", &numToAdd));  insert\_node(&headNode, numToAdd);  printf("Danh sách hiện tại sau khi chèn node mới: \n");  print\_list(headNode);  }  return 0;  }  void print\_list(struct *Node*\* *head*) {  struct *Node*\* currentNode = *head*;  /\* Lặp qua từng liên kết \*/  while (currentNode != NULL) {  printf("Giá trị: %d\n", currentNode->data);  currentNode = currentNode->next;  }  }  void insert\_node(struct *Node*\*\* *head*, int *nodeValue*) {  struct *Node*\* currentNode = malloc(sizeof \*currentNode);  currentNode->data = *nodeValue*;  currentNode->next = (\**head*);  \**head* = currentNode;  } |

**Giải thích cho việc chèn các nút**

Để hiểu cách chúng tôi thêm các nút ngay từ đầu, hãy xem xét các tình huống có thể xảy ra:

1. Danh sách trống, vì vậy chúng ta cần thêm một nút mới. Trong trường hợp đó, bộ nhớ của chúng ta trông như thế này khi HEAD là một con trỏ tới nút đầu tiên:

|  |
| --- |
| | HEAD | --> NULL |

Dòng currentNode->next = \*headNode; sẽ gán giá trị của currentNode->next là NULL vì headNode ban đầu bắt đầu với giá trị NULL.

Bây giờ, chúng tôi muốn đặt con trỏ nút đầu trỏ đến nút hiện tại của chúng tôi.

|  |
| --- |
| ----- -------------  |HEAD | --> |CURRENTNODE| --> NULL /\* Nút đầu trỏ tới nút hiện tại \*/  ----- ------------- |

Điều này được thực hiện với \*headNode = currentNode;

2. Danh sách đã được điền; chúng ta cần thêm một nút mới vào đầu. Để đơn giản, hãy bắt đầu với 1 nút:

|  |
| --- |
| ----- -----------  HEAD --> FIRST NODE --> NULL  ----- ----------- |

Với currentNode->next = \*headNode, cấu trúc dữ liệu sẽ như sau:

|  |
| --- |
| --------- ----- ---------------------  currentNode --> HEAD --> POINTER TO FIRST NODE --> NULL  --------- ----- --------------------- |

Mà, rõ ràng cần phải được thay đổi vì \*headNode nên trỏ đến currentNode.

|  |
| --- |
| ---- ----------- ---------------  HEAD -> currentNode --> NODE -> NULL  ---- ----------- --------------- |

Điều này được thực hiện với \*headNode = currentNode;

# ENUMERATIONS

## SIMPLE ENUMERATION (ENUMERATION ĐƠN GIẢN)

Một "enumeration" là một kiểu dữ liệu do người dùng định nghĩa, bao gồm các hằng số số nguyên và mỗi hằng số số nguyên được đặt tên. Từ khóa "enum" được sử dụng để định nghĩa kiểu dữ liệu được liệt kê.

Nếu bạn sử dụng "enum" thay vì "int" hoặc "string/char\*", bạn sẽ tăng tính kiểm tra thời gian biên dịch và tránh lỗi khi truyền các hằng số không hợp lệ, cũng như làm cho các giá trị hợp lệ sử dụng trở nên rõ ràng và dễ hiểu hơn.

**Ví dụ 1:**

|  |
| --- |
| enum *color*{ RED, GREEN, BLUE };  void printColor(enum *color* *chosenColor*)  {  const char \*color\_name = "Invalid color";  switch (*chosenColor*)  {  case RED:  color\_name = "RED";  break;    case GREEN:  color\_name = "GREEN";  break;  case BLUE:  color\_name = "BLUE";  break;  }  printf("%s\n", color\_name);  } |

Với một hàm main được định nghĩa như sau (ví dụ):

|  |
| --- |
| int main(){  enum *color* chosenColor;  printf("Enter a number between 0 and 2");  scanf("%d", (int\*)&chosenColor);  printColor(chosenColor);  return 0;  } |

**Ví dụ 2:**

(Ví dụ này sử dụng các bộ khởi tạo được chỉ định đã được chuẩn hóa từ C99.)

|  |
| --- |
| enum *week*{ MON, TUE, WED, THU, FRI, SAT, SUN };    static const char\* const dow[] = {  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };    void printDayOfWeek(enum *week* *day*)  {  printf("%s\n", dow[*day*]);  } |

Ví dụ tương tự sử dụng kiểm tra phạm vi:

|  |
| --- |
| enum *week*{ DOW\_INVALID = -1,  MON, TUE, WED, THU, FRI, SAT, SUN,  DOW\_MAX };    static const char\* const dow[] = {  [MON] = "Mon", [TUE] = "Tue", [WED] = "Wed",  [THU] = "Thu", [FRI] = "Fri", [SAT] = "Sat", [SUN] = "Sun" };    void printDayOfWeek(enum *week* *day*)  {  assert(*day* > DOW\_INVALID && *day* < DOW\_MAX);  printf("%s\n", dow[*day*]);  } |

## ENUMERATION CONSTANT WITHOUT TYPENAME (HẰNG SỐ ENUMERATION KHÔNG CẦN TÊN KIỂU)

Kiểu enumerations cũng có thể được khai báo mà không cần đặt tên cho chúng:

|  |
| --- |
| enum { buffersize = 256, };  static unsigned char buffer [buffersize] = { 0 }; |

Điều này cho phép chúng ta xác định các hằng số thời gian biên dịch kiểu int mà có thể như trong ví dụ này được sử dụng làm độ dài mảng.

## ENUMERATION WITH DUPLICATE VALUE (ENUMERATION VỚI GIÁ TRỊ TRÙNG LẶP)

Một enumeration không nhất thiết phải là duy nhất:

|  |
| --- |
| #include <stdlib.h> /\* cho EXIT\_SUCCESS \*/  #include <stdio.h> /\* cho printf() \*/  enum *Dupes*  {  Base, /\* Lấy giá trị 0 \*/  One, /\* Lấy giá trị của Base + 1 \*/  Two, /\* Lấy giá trị của One + 1 \*/  Negative = -1, /\* Gán giá trị là -1 \*/  AnotherZero /\* Lấy giá trị của Negative + 1 == 0, thật đáng tiếc \*/  };  int main(void)  {  printf("Base = %d\n", Base);  printf("One = %d\n", One);  printf("Two = %d\n", Two);  printf("Negative = %d\n", Negative);  printf("AnotherZero = %d\n", AnotherZero);  return EXIT\_SUCCESS;  } |

Kết quả:

|  |
| --- |
| Base = 0  One = 1  Two = 2  Negative = -1  AnotherZero = 0 |

## TYPEDEF ENUM

Có một số khả năng và quy ước để đặt tên cho một enumeration. Đầu tiên là sử dụng một thẻ (tag) ngay sau từ khóa "**enum**".

|  |
| --- |
| enum *color*  {  RED,  GREEN,  BLUE  }; |

Kiểu liệt kê này phải luôn được sử dụng cùng với từ khóa "enum" và thẻ tương ứng như sau:

|  |
| --- |
| enum *color* chosenColor = RED; |

Nếu chúng ta sử dụng "**typedef**" trực tiếp khi khai báo **enum**, chúng ta có thể bỏ qua thẻ và sau đó sử dụng kiểu mà không có từ khóa "enum":

|  |
| --- |
| typedef enum  {  RED,  GREEN,  BLUE  } *color*; |

Sau đó, chúng ta có thể sử dụng "color" mà không cần từ khóa "enum":

|  |
| --- |
| *color* chosenColor = RED; |

Tuy nhiên, trong trường hợp này, chúng ta không thể sử dụng "enum color" vì chúng ta không sử dụng thẻ trong định nghĩa. Một quy ước phổ biến là sử dụng cả hai, để cùng một tên có thể được sử dụng với hoặc không có từ khóa "enum". Điều này có lợi thế đặc biệt là tương thích với C++.

|  |
| --- |
| enum *color* /\* như ví dụ đầu tiên \*/  {  RED,  GREEN,  BLUE  };  typedef enum *color* *color*; /\* cũng là một typedef với cùng một thay đổi \*/  *color* chosenColor = RED;  enum *color* defaultColor = BLUE; |

Hàm

|  |
| --- |
| void printColor()  {  if (chosenColor == RED)  {  printf("RED\n");  }  else if (chosenColor == GREEN)  {  printf("GREEN\n");  }  else if (chosenColor == BLUE)  {  printf("BLUE\n");  }  } |

Để biết thêm về "typedef", xem phần "Typedef".

# STRUCTS (CẤU TRÚC)

Cấu trúc (Structs) cung cấp một cách để nhóm một tập hợp các biến liên quan thuộc nhiều loại khác nhau vào một đơn vị bộ nhớ. Toàn bộ cấu trúc có thể được tham chiếu bằng một tên hoặc con trỏ duy nhất; các thành viên cấu trúc cũng có thể được truy cập riêng lẻ. Các cấu trúc có thể được truyền đến các hàm và được trả về từ các hàm. Chúng được định nghĩa bằng cách sử dụng từ khóa **struct**.

## FLEXIBLE ARRAY MEMBERS (THÀNH PHẦN MẢNG LINH HOẠT)

### Khai báo kiểu dữ liệu

Một cấu trúc có ít nhất một thành viên có thể chứa thêm một thành viên mảng có độ dài không xác định ở cuối cấu trúc. Đây được gọi là thành viên mảng linh hoạt:

|  |
| --- |
| struct *ex1*  {  *size\_t* foo;  int flex[]  };  struct *ex2\_header*  {  int foo;  char bar;  };  struct *ex2*  {  struct *ex2\_header*;  int flex;  };  /\* Hợp nhất cấu trúc ex2\_header và ex2. \*/  struct *ex3*  {  int foo;  char bar  int flex[];  }; |

### 13.1.2 Ảnh hưởng đến kích thước và phần đệm

Thành viên mảng linh hoạt được coi là không có kích thước khi tính toán kích thước của cấu trúc, mặc dù phần đệm giữa thành viên đó và thành viên trước đó của cấu trúc vẫn có thể tồn tại:

|  |
| --- |
| /\* In ra "8,8" trên máy tính của tôi, vì vậy không có phần trống. \*/  printf("%zu,%zu\n", sizeof(size\_t), sizeof(struct ex1));  /\* Cũng in ra "8,8" trên máy tính của tôi, nên không có phần trống trong cấu trúc ex2 chính nó. \*/  printf("%zu,%zu\n", sizeof(struct ex2\_header), sizeof(struct ex2));  /\* In ra "5,8" trên máy tính của tôi, nên có 3 byte phần trống. \*/  printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3)); |

Thành viên mảng linh hoạt (Flexible Array Member) được coi là có kiểu mảng không hoàn chỉnh, do đó không thể tính kích thước của nó bằng cách sử dụng "**sizeof**".

### 13.1.3 Sử dụng

Bạn có thể khai báo và khởi tạo một đối tượng có kiểu cấu trúc chứa một thành viên mảng linh hoạt (Flexible Array Member), nhưng bạn không được thử khởi tạo thành viên mảng linh hoạt (Flexible Array Member) vì nó được xử lý như không tồn tại. Thử làm điều này sẽ bị cấm và gây ra lỗi biên dịch.

Tương tự, bạn không nên cố gắng gán giá trị cho bất kỳ phần tử nào của thành viên mảng linh hoạt (Flexible Array Member) khi khai báo cấu trúc theo cách này vì có thể không có đủ phần trống ở cuối cấu trúc để chứa bất kỳ đối tượng nào cần thiết cho thành viên mảng linh hoạt (Flexible Array Member). Trình biên dịch không nhất thiết phải ngăn bạn làm điều này, do đó có thể dẫn đến hành vi không xác định.

|  |
| --- |
| /\* Không hợp lệ: không thể khởi tạo thành viên mảng linh hoạt (Flexible Array Member) \*/  struct *ex1* e1 = {1, {2, 3}};  /\* Không hợp lệ: hdr={foo=1, bar=2} OK, nhưng không thể khởi tạo thành viên mảng linh hoạt (Flexible Array Member) \*/  struct *ex2* e2 = {{1, 2}, {3}};  /\* Hợp lệ: khởi tạo thành viên foo=1, bar=2 \*/  struct *ex3* e3 = {1, 2};  e1.flex[0] = 3; /\* Hành vi không xác định, trong trường hợp của tôi \*/  e3.flex[0] = 2; /\* Hành vi không xác định lại \*/  e2.flex[0] = e3.flex[0]; /\* Hành vi không xác định \*/ |

Thay vào đó, bạn có thể chọn sử dụng malloc, calloc hoặc realloc để cấp phát cấu trúc có lưu trữ bổ sung và sau đó giải phóng nó, cho phép bạn sử dụng thành viên mảng linh hoạt (Flexible Array Member) theo ý muốn:

|  |
| --- |
| /\* Hợp lệ: cấp phát một đối tượng có kiểu cấu trúc ex1 cùng với một mảng gồm 2 số nguyên \*/  struct *ex1* \*pe1 = malloc(sizeof(\*pe1) + 2 \* sizeof(pe1->flex[0]));  /\* Hợp lệ: cấp phát một đối tượng có kiểu cấu trúc ex2 cùng với một mảng gồm 4 số nguyên \*/  struct *ex2* \*pe2 = malloc(sizeof(struct *ex2*) + sizeof(int[4]));  /\* Hợp lệ: cấp phát 5 đối tượng kiểu cấu trúc ex3 cùng với một mảng gồm 3 số nguyên cho mỗi đối tượng \*/  struct *ex3* \*pe3 = malloc(5 \* (sizeof(\*pe3) + sizeof(int[3])));  pe1->flex[0] = 3; /\* có hiệu lực \*/  pe3[0]->flex[0] = pe1->flex[0]; /\* có hiệu lực \*/ |

### 13.1.4 Struct hack

Phiên bản C trước C99 không hỗ trợ "flexible array members," bạn có thể sử dụng một kỹ thuật được gọi là "struct hack" để thay thế. Trong"struct hack," bạn sẽ khai báo một mảng có kích thước là 1 trong cấu trúc để thay thế "flexible array members."

|  |
| --- |
| struct *ex1*  {  *size\_t* foo;  int flex[1];  }; |

Tuy nhiên, điều này sẽ ảnh hưởng đến kích thước của cấu trúc, không giống như một thành viên mảng linh hoạt thực sự:

|  |
| --- |
| /\* In "8,4,16" trên máy của tôi, biểu thị rằng có 4 byte đệm. \*/  printf("%d,%d,%d\n", (int)sizeof(size\_t), (int)sizeof(int[1]), (int)sizeof(struct ex1)); |

Để sử dụng thành viên "flex" như một thành viên mảng linh hoạt, bạn sẽ cấp phát bộ nhớ cho nó bằng cách sử dụng malloc như đã được trình bày ở trên, nhưng thay vì sử dụng **sizeof**(\*pe1) (hoặc **sizeof(struct** ex1)), bạn sẽ thay thế bằng offsetof(**struct** ex1, flex) hoặc biểu thức dài hơn, không phụ thuộc vào kiểu dữ liệu **sizeof**(\*pe1) - **sizeof**(pe1->flex). Hoặc có thể trừ đi 1 từ độ dài mong muốn của mảng "linh hoạt" vì nó đã được bao gồm trong kích thước của cấu trúc, giả sử độ dài mong muốn là lớn hơn 0. Cùng logic này có thể được áp dụng vào các ví dụ sử dụng khác.

### 13.1.5 Khả năng tương thích

Nếu muốn tương thích với các trình biên dịch không hỗ trợ các thành viên mảng linh hoạt, bạn có thể sử dụng macro được xác định như FLEXMEMB\_SIZE bên dưới:

|  |
| --- |
| #if \_\_STDC\_VERSION\_\_ < 199901L  #define FLEXMEMB\_SIZE 1  #else  #define FLEXMEMB\_SIZE /\* nothing \*/  #endif  struct *ex1*  {  *size\_t* foo;  int flex[FLEXMEMB\_SIZE];  }; |

Khi cấp phát các đối tượng, bạn nên sử dụng offsetof(struct ex1, flex) để tham chiếu đến kích thước của cấu trúc (loại trừ thành viên mảng linh hoạt) vì nó là biểu thức duy nhất giữ nguyên tính nhất quán giữa các trình biên dịch hỗ trợ thành viên mảng linh hoạt và các trình biên dịch không hỗ trợ:

|  |
| --- |
| struct *ex1* \*pe10 = malloc(offsetof(struct *ex1*, flex) + n \* sizeof(pe10->flex[0])); |

Một phương pháp thay thế khác là sử dụng tiền xử lý để trừ 1 từ độ dài được chỉ định. Tuy nhiên, do tiềm năng tăng cao về không nhất quán và lỗi của con người trong hình thức này, tôi đã di chuyển logic vào một hàm riêng biệt:

|  |
| --- |
| struct *ex1* \*ex1\_alloc(*size\_t* *n*)  {  struct *ex1* tmp;  #if \_\_STDC\_VERSION\_\_ < 199901L  if (n != 0)  n--;  #endif  return malloc(sizeof(tmp) + *n* \* sizeof(tmp.flex[0]));  }  ...  /\* cấp phát một đối tượng ex1 với mảng "flex" có độ dài là 3 \*/  struct ex1 \*pe1 = ex1\_alloc(3); |

## PHẦN 13.2 TYPEDEF STRUCTS

Kết hợp từ khóa typedef với struct có thể làm mã nguồn rõ ràng hơn. Ví dụ:

|  |
| --- |
| typedef struct  {  int x, y;  } *Point*; |

Thay vì:

|  |
| --- |
| struct *Point*  {  int x, y;  }; |

có thể được khai báo như sau:

|  |
| --- |
| *Point* point; |

Thay vì

|  |
| --- |
| struct *Point* point; |

Một cách tốt hơn là sử dụng cách khai báo sau:

|  |
| --- |
| typedef struct *Point* *Point*;  struct *Point*  {  int x, y;  }; |

để có lợi ích của cả hai cách định nghĩa point. Khai báo như vậy là tiện lợi nhất nếu bạn đã học C++ trước, nơi bạn có thể bỏ qua từ khóa struct nếu tên không gây nhiễu.

typedef cho các tên struct có thể xung đột với các bộ nhớ đệm khác trong chương trình. Một số người coi điều này là một nhược điểm, nhưng đối với hầu hết mọi người, khi có một struct và một bộ nhớ đệm khác có cùng tên sẽ gây khó chịu.

Một điểm nổi tiếng là hàm stat của POSIX:

|  |
| --- |
| int stat(const char \**pathname*, struct *stat* \**buf*); |

trong đó bạn thấy hàm stat có một đối số là struct stat.

Các struct được định nghĩa bằng typedef mà không có tên thẻ luôn yêu cầu rằng toàn bộ khai báo struct phải được hiển thị trong mã sử dụng nó. Toàn bộ khai báo struct phải được đặt trong một tệp header.

Coi như:

|  |
| --- |
| #include "bar.h"  struct *foo*  {  bar \*aBar;  }; |

Khi sử dụng **typedef** cho **struct** mà không có tên thẻ, tệp **bar.h** luôn phải bao gồm toàn bộ định nghĩa của cấu trúc **bar**. Nếu chúng ta sử dụng typedef như sau trong tệp bar.h:

|  |
| --- |
| typedef struct *bar* *bar*; |

trong **bar.h**, các chi tiết của cấu trúc bar có thể được ẩn đi.

Xem thêm về Typedef

## PHẦN 13.3 POINTERS TO STRUCTS (CON TRỎ TỚI CẤU TRÚC)

Khi bạn có một biến chứa một cấu trúc, bạn có thể truy cập các trường của nó bằng toán tử chấm (.). Tuy nhiên, nếu bạn có một con trỏ tới một cấu trúc, điều này sẽ không hoạt động. Bạn phải sử dụng toán tử mũi tên (->) để truy cập các trường của nó. Dưới đây là một ví dụ về cách triển khai cực kỳ đơn giản (một số người có thể gọi là "đơn giản và kém chất lượng") của một ngăn xếp sử dụng con trỏ tới cấu trúc và thể hiện toán tử mũi tên.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  /\* cấu trúc \*/  struct *stack*  {  struct *node* \*top;  int size;  };  struct *node*  {  int data;  struct *node* \*next;  };  /\* khai báo hàm \*/  int push(int, struct *stack* \*);  int pop(struct *stack* \*);  void destroy(struct *stack* \*);  int main(void)  {  int result = EXIT\_SUCCESS;  *size\_t* i;  /\* cấp phát bộ nhớ cho một struct stack và ghi nhận con trỏ của nó \*/  struct *stack* \*stack = malloc(sizeof \*stack);  if (NULL == stack)  {  perror("malloc() failed");  return EXIT\_FAILURE;  }  /\* khởi tạo ngăn xếp \*/  stack->top = NULL;  stack->size = 0;  /\* đẩy 10 số nguyên vào ngăn xếp \*/  {  int data = 0;  for (i = 0; i < 10; i++)  {  printf("Pushing: %d\n", data);  if (-1 == push(data, stack))  {  perror("push() failed");  result = EXIT\_FAILURE;  break;  }  ++data;  }  }  if (EXIT\_SUCCESS == result)  {  /\* lấy ra 5 số nguyên từ ngăn xếp \*/  for (i = 0; i < 5; i++)  {  printf("Popped: %i\n", pop(stack));  }  }  /\* hủy ngăn xếp \*/  destroy(stack);  return result;  }  /\* Đẩy một giá trị vào ngăn xếp. \*/  /\* Trả về 0 nếu thành công và -1 nếu thất bại. \*/  int push(int *data*, struct *stack* \**stack*)  {  int result = 0;  /\* cấp phát bộ nhớ cho nút mới \*/  struct *node* \*new\_node = malloc(sizeof \*new\_node);  if (NULL == new\_node)  {  result = -1;  }  else  {  new\_node->data = *data*;  new\_node->next = *stack*->top;  *stack*->top = new\_node;  *stack*->size++;  }  return result;  }  /\* Lấy ra một giá trị từ ngăn xếp. \*/  /\* Trả về giá trị lấy ra từ ngăn xếp \*/  int pop(struct *stack* \**stack*)  {  struct *node* \*top = *stack*->top;  int data = top->data;  *stack*->top = top->next;  *stack*->size--;  free(top);  return data;  }  /\* hủy ngăn xếp \*/  void destroy(struct *stack* \**stack*)  {  /\* giải phóng tất cả con trỏ \*/  while (*stack*->top != NULL)  {  pop(*stack*);  }  } |

## PHẦN 13.4 PASSING STRUCTS TO FUNCTIONS (TRUYỀN STRUCTS VÀO HÀM)

Trong C, tất cả các đối số được truyền vào các hàm theo giá trị, bao gồm cả các cấu trúc. Đối với các cấu trúc nhỏ, điều này rất tốt vì không có chi phí từ việc truy cập dữ liệu thông qua con trỏ. Tuy nhiên, điều này cũng khiến việc truyền ngẫu nhiên một cấu trúc lớn dễ dàng, dẫn đến hiệu suất kém, đặc biệt nếu lập trình viên đã sử dụng các ngôn ngữ khác nơi đối số được truyền theo tham chiếu.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  /\* cấu trúc \*/  struct *coordinates*  {  int x;  int y;  int z;  };  /\* Hàm trả về cấu trúc nhỏ bằng giá trị, rất nhanh \*/  struct *coordinates* move(struct *coordinates* *position*, struct *coordinates* *movement*)  {  *position*.x += *movement*.x;  *position*.y += *movement*.y;  *position*.z += *movement*.z;  return *position*;  }  /\* Cấu trúc rất lớn \*/  struct *lotsOfData*  {  int param1;  char param2[80000];  };  /\* Hàm trả về cấu trúc lớn bằng giá trị, rất chậm! \*/  /\* Với kích thước lớn của cấu trúc này, có thể gây tràn ngăn xếp (stack overflow) \*/  struct *lotsOfData* doubleParam1(struct *lotsOfData* *value*)  {  *value*.param1 \*= 2;  return *value*;  }  /\* Hàm trả về cấu trúc lớn bằng con trỏ thay vì giá trị, tương đối nhanh \*/  void doubleParam1ByPtr(struct *lotsOfData* \**value*)  {  *value*->param1 \*= 2;  } |

## PHẦN 13.5 OBJECT- BASED PROGRAMMING USING STRUCTS (LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG SỬ DỤNG STRUCTS)

Cấu trúc có thể được sử dụng để triển khai mã theo cách lập trình hướng đối tượng. Một cấu trúc tương tự như một lớp, nhưng thiếu các hàm mà thông thường cũng là một phần của một lớp, chúng ta có thể thêm chúng vào dưới dạng biến thành viên con trỏ hàm. Để giữ nguyên ví dụ về tọa độ của chúng ta:

|  |
| --- |
| /\* coordinates.h \*/  typedef struct *coordinate\_s*  {  /\* Con trỏ tới các hàm phương thức \*/  void (\*setx)(*coordinate* \*this, int x);  void (\*sety)(*coordinate* \*this, int y);  void (\*print)(*coordinate* \*this);  /\* Dữ liệu \*/  int x;  int y;  } *coordinate*;  /\* Constructor \*/  *coordinate* \*coordinate\_create(void);  /\* Destructor \*/  void coordinate\_destroy(*coordinate* \**this*); |

Và sau đó tạo tệp thực hiện C:

|  |
| --- |
| /\* coordinates.c \*/  #include "coordinates.h"  #include <stdio.h>  #include <stdlib.h>  /\* Constructor \*/  coordinate \*coordinate\_create(void)  {  coordinate \*c = malloc(sizeof(\*c));  if (c != 0)  {  c->setx = &coordinate\_setx;  c->sety = &coordinate\_sety;  c->print = &coordinate\_print;  c->x = 0;  c->y = 0;  }  return c;  }  /\* Destructor \*/  void coordinate\_destroy(coordinate \**this*)  {  if (*this* != NULL)  {  free(*this*);  }  }  /\* Phương thức \*/  static void coordinate\_setx(coordinate \**this*, int *x*)  {  if (*this* != NULL)  {  *this*->x = *x*;  }  }  static void coordinate\_sety(coordinate \**this*, int *y*)  {  if (*this* != NULL)  {  *this*->y = *y*;  }  }  static void coordinate\_print(coordinate \**this*)  {  if (*this* != NULL)  {  printf("Coordinate: (%i, %i)\n", *this*->x, *this*->y);  }  else  {  printf("NULL pointer exception!\n");  }  } |

Một ví dụ về việc sử dụng lớp tọa độ của chúng ta sẽ là:

|  |
| --- |
| /\* main.c \*/  #include "coordinates.h"  #include <stddef.h>  int main(void)  {  /\* Tạo và khởi tạo các con trỏ tới các đối tượng tọa độ \*/  coordinate \*c1 = coordinate\_create();  coordinate \*c2 = coordinate\_create();  /\* Bây giờ chúng ta có thể sử dụng các đối tượng của chúng ta bằng cách sử dụng các phương thức và truyền đối tượng như là tham số \*/  c1->setx(c1, 1);  c1->sety(c1, 2);  c2->setx(c2, 3);  c2->sety(c2, 4);  c1->print(c1);  c2->print(c2);  /\* Sau khi sử dụng các đối tượng của chúng ta, chúng ta hủy chúng bằng cách sử dụng hàm "hủy" của chúng ta \*/  coordinate\_destroy(c1);  c1 = NULL;  coordinate\_destroy(c2);  c2 = NULL;  return 0;  } |

## PHẦN 13.6 SIMPLE DATA STRUCTURES (CÁC CẤU TRÚC DỮ LIỆU ĐƠN GIẢN)

Cấu trúc dữ liệu kiểu struct là cách hữu ích để đóng gói các dữ liệu liên quan và cho phép chúng hoạt động như một biến duy nhất.

Khai báo một cấu trúc đơn giản chứa hai trường kiểu int:

|  |
| --- |
| struct *point*  {  int x;  int y;  }; |

x và y được gọi là các thành viên (hoặc trường) của cấu trúc point.

Định nghĩa và sử dụng các cấu trúc:

|  |
| --- |
| struct *point* p; // khai báo p là một cấu trúc point  p.x = 5; // gán các biến thành viên của p  p.y = 3; |

Cấu trúc có thể được khởi tạo tại định nghĩa. Đoạn mã trên tương đương với:

|  |
| --- |
| struct *point* p = {5, 3}; |

Cấu trúc cũng có thể được khởi tạo bằng cách sử dụng các trình khởi tạo được chỉ định (designated initializers).

Truy cập các trường cũng được thực hiện bằng cách sử dụng toán tử chấm (.)

|  |
| --- |
| printf("point is (x = %d, y = %d)", p.x, p.y); |

# STANDARD MATH (TOÁN HỌC TIÊU CHUẨN)

## CÁC HÀM MŨ – POW(), POWF(), POWL()

Đoạn mã ví dụ sau tính tổng của dãy 1 + 4(3 + 3^2 + 3^3 + 3^4 + ... + 3^N) sử dụng các hàm gia đình pow() trong thư viện toán học tiêu chuẩn.

|  |
| --- |
| #include <stdio.h>  #include <math.h>  #include <errno.h>  #include <fenv.h>  int main()  {  double pwr, sum = 0;  int i, n;  printf("\n1 + 4(3 + 3^2 + 3^3 + 3^4 + ... + 3^N) = ?\nEnter N:");  scanf("%d", &n);  if (n <= 0)  {  printf("Invalid power N=%d", n);  return -1;  }  for (i = 0; i < n + 1; i++)  {  errno = 0;  feclearexcept(FE\_ALL\_EXCEPT);  pwr = powl(3, i);  if (fetestexcept(FE\_INVALID | FE\_DIVBYZERO | FE\_OVERFLOW | FE\_UNDERFLOW))  {  perror("Math Error");  }  sum += i ? pwr : 0;  printf("N = %d\tS = %g\n", i, 1 + 4 \* sum);  }  return 0;  } |

Kết quả:

|  |
| --- |
| 1 + 4(3 + 3^2 + 3^3 + 3^4 + ... + 3^N) = ?  Enter N:10  N = 0 S = 1  N = 1 S = 13  N = 2 S = 49  N = 3 S = 157  N = 4 S = 481  N = 5 S = 1453  N = 6 S = 4369  N = 7 S = 13117  N = 8 S = 39361  N = 9 S = 118093  N = 10 S = 354289 |

## DOUBLE PRECISION FLOATING- POINT REMAINDER FMOD() (PHẦN DƯ SỐ THỰC VỚI ĐỘ CHÍNH XÁC KÉP FMOD())

Hàm fmod() trả về phần dư số thực độ chính xác kép của phép chia x/y. Giá trị trả về có cùng dấu với x.

|  |
| --- |
| #include <math.h> /\* for fmod() \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {  double x = 10.0;  double y = 5.1;  double modulus = fmod(x, y);  printf("%lf\n", modulus); /\* f tương đương với lf \*/  return 0;  } |

Kết quả:

|  |
| --- |
| 4.900000 |

**Lưu ý:** Sử dụng hàm này cẩn thận, vì nó có thể trả về các giá trị không mong đợi do hoạt động của các giá trị số thực.

|  |
| --- |
| #include <math.h>  #include <stdio.h>  int main(void)  {  printf("%f\n", fmod(1, 0.1));  printf("%19.17f\n", fmod(1, 0.1));  return 0;  } |

Kết quả:

|  |
| --- |
| 0.100000  0.09999999999999995 |

## SINGLE PRECISION AND LONG DOUBLE PRECISION FLOATING- POINT REMAINDER: FMODF(), FMODL()

Hàm fmodf() và fmodl() trả về phần dư số thực độ chính xác đơn và đôi của phép chia x/y. Giá trị trả về có cùng dấu với x.

Số thực độ chính xác đơn (Single Precision):

|  |
| --- |
| #include <math.h> /\* for fmodf() \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {  float x = 10.0;  float y = 5.1;  float modulus = fmodf(x, y);  printf("%f\n", modulus); /\* Lf sẽ hoạt động tốt khi modulus được thăng cấp thành kiểu double. \*/  return 0;  } |

Kết quả:

|  |
| --- |
| 4.900000 |

Số thực độ chính xác kép (Double Precision):

|  |
| --- |
| #include <math.h> /\* for fmodl() \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {  long double x = 10.0;  long double y = 5.1;  long double modulus = fmodl(x, y);  printf("%Lf\n", modulus); /\* Lf dùng cho số thực độ chính xác kép (long double). \*/  return 0;  } |

Kết quả:

|  |
| --- |
| 4.90000 |

# CÁC LỆNH LẶP/ VÒNG LẶP: FOR, WHILE, DO WHILE

## PHẦN 15.1 VÒNG LẶP FOR

Để thực hiện một khối mã lệnh nhiều lần, ta sử dụng vòng lặp (loop). Vòng lặp **for** được sử dụng khi một khối mã lệnh cần được thực thi một số lần cụ thể. Ví dụ, để điền vào một mảng có kích thước n với đầu vào từ người dùng, chúng ta cần thực hiện lệnh **scanf()** **n** lần.

|  |
| --- |
| #include <stddef.h> // for size\_t  int array[10]; // mảng gồm 10 số nguyên  for (size\_t i = 0; i < 10; i++) // i bắt đầu từ 0 và kết thúc ở 9  {  scanf("%d", &array[i]);  } |

Theo cách này, lệnh gọi scanf() được thực hiện n lần (10 lần trong ví dụ của chúng ta), nhưng được viết một lần duy nhất.

Ở đây, biến i là chỉ mục vòng lặp và nó nên được khai báo như trình bày. Kiểu size\_t (kiểu kích thước) nên được sử dụng cho mọi thứ đếm hoặc lặp qua các đối tượng dữ liệu.

Cách khai báo biến bên trong vòng lặp for chỉ có sẵn cho các trình biên dịch đã được cập nhật lên tiêu chuẩn C99. Nếu vì một lý do nào đó bạn vẫn phải sử dụng trình biên dịch cũ hơn, bạn có thể khai báo chỉ mục vòng lặp trước vòng lặp for:

|  |
| --- |
| #include <stddef.h> /\* for size\_t \*/  *size\_t* i;  int array[10]; /\* mảng gồm 10 số nguyên \*/  for (i = 0; i < 10; i++) /\* i bắt đầu từ 0 và kết thúc ở 9 \*/  {  scanf("%d", &array[i]);  } |

## PHẦN 15.2 LOOP UNROLLING AND DUFF’S DEVICE

Đôi khi, vòng lặp thông thường không thể được hoàn toàn chứa trong thân vòng lặp. Điều này là vì, vòng lặp cần được chuẩn bị bởi một số lệnh B. Sau đó, vòng lặp bắt đầu với một số lệnh A, sau đó tiếp tục với B trước khi lặp lại.

|  |
| --- |
| do\_B();  while (condition) {  do\_A();  do\_B();  } |

Để tránh vấn đề coppy/paste tiềm ẩn khi lặp lại B hai lần trong mã, **Duff's Device** có thể được áp dụng để bắt đầu vòng lặp từ giữa phần thân của while, bằng cách sử dụng một câu lệnh switch và cách thức chuyển hướng dưới đây.

|  |
| --- |
| switch (true) while (condition) {  case false: do\_A(); /\* FALL THROUGH \*/  default: do\_B(); /\* FALL THROUGH \*/  } |

Thực tế, Duff's Device được phát minh để thực hiện giải thuật gỡ vòng lặp. Giả sử bạn áp dụng một mặt nạ vào một khối bộ nhớ, trong đó n là một kiểu số nguyên có giá trị dương.

|  |
| --- |
| do {  \*ptr++ ^= mask;  } while (--n > 0); |

Nếu n luôn chia hết cho 4, bạn có thể dễ dàng gỡ vòng lặp này như sau:

|  |
| --- |
| do {  \*ptr++ ^= mask;  \*ptr++ ^= mask;  \*ptr++ ^= mask;  \*ptr++ ^= mask;  } while ((n -= 4) > 0); |

Nhưng, với Duff's Device, mã có thể tuân theo giải thuật gỡ vòng lặp này bằng cách nhảy vào vị trí đúng giữa vòng lặp nếu n không chia hết cho 4.

|  |
| --- |
| switch (n % 4) do {  case 0: \*ptr++ ^= mask; /\* FALL THROUGH \*/  case 3: \*ptr++ ^= mask; /\* FALL THROUGH \*/  case 2: \*ptr++ ^= mask; /\* FALL THROUGH \*/  case 1: \*ptr++ ^= mask; /\* FALL THROUGH \*/  } while ((n -= 4) > 0); |

Loại bỏ vòng lặp thủ công như vậy hiếm khi cần thiết với các trình biên dịch hiện đại, vì trình tối ưu hóa của trình biên dịch có thể thực hiện gỡ vòng lặp thay mặt người lập trình.

## PHẦN 15.3 VÒNG LẶP WHILE

Vòng lặp while được sử dụng để thực thi một đoạn mã khi điều kiện là đúng. Vòng lặp while được sử dụng khi một khối mã cần được thực thi một số lần có thể thay đổi. Ví dụ mã dưới đây nhận đầu vào từ người dùng, miễn là người dùng nhập các số khác không. Nếu người dùng nhập số 0, điều kiện của vòng lặp while không còn đúng nữa, vòng lặp sẽ kết thúc và tiếp tục thực thi các mã tiếp theo:

|  |
| --- |
| int num = 1;  while (num != 0)  {  scanf("%d", &num);  } |

## PHẦN 15.4 VÒNG LẶP DO – WHILE

Khác với vòng lặp for và while, vòng lặp do-while kiểm tra điều kiện sau khi thực thi phần thân của vòng lặp, điều này có nghĩa là khối mã trong do sẽ được thực thi ít nhất một lần và sau đó kiểm tra điều kiện của while ở cuối khối mã. Điều này đồng nghĩa với việc vòng lặp do-while luôn chạy ít nhất một lần.

Ví dụ vòng lặp do-while sau sẽ lấy các số từ người dùng cho đến khi tổng các giá trị này lớn hơn hoặc bằng 50:

|  |
| --- |
| int num, sum;  num = sum = 0;  do  {  scanf("%d", &num);  sum += num;  } while (sum < 50); |

Vòng lặp do-while có thể gặp ít hơn trong hầu hết các phong cách lập trình.

## PHẦN 15.5 CẤU TRÚC VÀ LUỒNG ĐIỀU KHIỂN TRONG VÒNG LẶP FOR

|  |
| --- |
| for([khai báo hoặc biểu thức 1]; [biểu thức 2]; [biểu thức 3]){  /\*Phần thân của vòng lặp\*/  } |

Trong một vòng lặp for, điều kiện lặp có ba biểu thức, tất cả đều là tùy chọn.

* Biểu thức đầu tiên, được gọi là khai báo hoặc biểu thức "declaration-or-expression", khởi tạo vòng lặp. Nó chỉ được thực thi một lần duy nhất ở đầu của vòng lặp.

Phiên bản ≥ C99:

Nó có thể là một khai báo và khởi tạo của biến vòng lặp, hoặc là một biểu thức tổng quát. Nếu nó là một khai báo, phạm vi của biến được khai báo được **giới hạn** bởi câu lệnh for.

Phiên bản < C99:

Các phiên bản C cũ chỉ cho phép sử dụng biểu thức ở đây, và khai báo biến vòng lặp phải được đặt trước vòng lặp for.

* Biểu thức thứ hai, được gọi là biểu thức 2"expression2", là điều kiện kiểm tra. Nó được thực thi lần đầu tiên sau khi khởi tạo. Nếu điều kiện **đúng**, thì điều khiển vào phần thân của vòng lặp. Nếu **không đúng,** nó chuyển đến bên ngoài phần thân của vòng lặp ở cuối vòng lặp. Sau đó, điều kiện này được kiểm tra sau mỗi thực thi của phần thân cũng như biểu thức cập nhật. Khi điều kiện đúng, điều khiển quay lại đầu phần thân của vòng lặp. Điều kiện này thường được dùng để kiểm tra số lần thực thi của phần thân vòng lặp. Đây là cách chính để thoát khỏi vòng lặp, cách khác là sử dụng các câu lệnh nhảy (jump statements).
* Biểu thức thứ ba, được gọi là biểu thức 3"expression3", là **câu lệnh cập nhật**. Nó được thực thi sau mỗi thực thi của phần thân của vòng lặp. Thường thì nó được sử dụng để tăng giá trị của biến lưu số lần thực thi của phần thân vòng lặp, và biến này được gọi là "**iterator**".

Mỗi lần thực thi của phần thân vòng lặp được gọi là một "**iteration**".

Ví dụ:

Phiên bản ≥ C99

|  |
| --- |
| for (int i = 0; i < 10; i++)  {  printf("%d", i);  } |

Kết quả

|  |
| --- |
| 0123456789 |

Trong ví dụ trên, trước tiên i = 0 được thực thi, khởi tạo i. Sau đó, điều kiện i < 10 được kiểm tra, nó đánh giá là đúng. Điều khiển vào phần thân của vòng lặp và giá trị của i được in ra. Sau đó, điều khiển chuyển đến i++, cập nhật giá trị của i từ 0 lên 1. Sau đó, điều kiện lại được kiểm tra và quá trình tiếp tục. Điều này tiếp tục cho đến khi giá trị của i trở thành 10. Sau đó, điều kiện i < 10 đánh giá sai, sau đó điều khiển di chuyển ra khỏi vòng lặp.

## PHẦN 15.6 VÒNG LẶP VÔ HẠN

Một vòng lặp được gọi là vòng lặp vô hạn nếu điều khiển vào nhưng không bao giờ rời khỏi phần thân của vòng lặp. Điều này xảy ra khi điều kiện kiểm tra của vòng lặp không bao giờ đánh giá là **sai**.

Ví dụ:

Phiên bản ≥ C99

|  |
| --- |
| for (int i = 0; i >= 0; )  {  /\* phần thân của vòng lặp, nơi i không bị thay đổi\*/  } |

Trong ví dụ trên, biến i, bộ đếm, được khởi tạo thành 0. Điều kiện kiểm tra ban đầu đúng. Tuy nhiên, i không được thay đổi ở bất kỳ nơi nào trong phần thân và biểu thức cập nhật không có gì. Do đó, i sẽ giữ nguyên là 0 và điều kiện kiểm tra sẽ không bao giờ đánh giá sai, dẫn đến một vòng lặp vô hạn.

Giả định rằng không có câu lệnh nhảy, một cách khác mà một vòng lặp vô hạn có thể được tạo ra là bằng cách giữ cho điều kiện luôn luôn đúng:

|  |
| --- |
| while (true)  {  /\* phần thân của vòng lặp \*/  } |

Trong vòng lặp for, câu lệnh điều kiện là tùy chọn. Trong trường hợp này, điều kiện luôn luôn đúng, dẫn đến một vòng lặp vô hạn.

|  |
| --- |
| for (;;)  {  /\* phần thân của vòng lặp \*/  } |

Tuy nhiên, trong một số trường hợp, điều kiện có thể được giữ đúng một cách có ý định, với mục đích thoát khỏi vòng lặp bằng một câu lệnh nhảy như **break**.

|  |
| --- |
| while (true)  {  /\* các câu lệnh \*/  if (condition)  {  /\* các câu lệnh khác \*/  break;  }  } |

# SELECTION STATEMENTS (CÂU LỆNH LỰA CHỌN)

## PHẦN 16.1 CÂU LỆNH IF()

Một trong những cách đơn giản nhất để điều khiển luồng chương trình là sử dụng câu lệnh if (). Có thể quyết định liệu một khối mã có được thực thi hay không dựa vào câu lệnh này.

Cú pháp cho câu lệnh if trong C có thể như sau:

|  |
| --- |
| if (điều\_kiện)  {  các\_câu\_lệnh; /\* được thực thi khi điều kiện là true \*/  } |

Ví dụ:

|  |
| --- |
| if (a > 1) {  puts("a lớn hơn 1");  } |

Ở đây, a > 1 là một điều kiện phải đánh giá là **true**(đúng) để thực thi các câu lệnh bên trong khối if. Trong ví dụ này, "a lớn hơn 1" chỉ được in ra màn hình nếu a > 1 là **true**(đúng).

Câu lệnh if có thể bỏ qua cặp dấu ngoặc nhọn { and } nếu chỉ có một câu lệnh trong khối. Ví dụ trên có thể viết lại thành:

|  |
| --- |
| if (a > 1)  puts("a lớn hơn 1"); |

Tuy nhiên, để thực thi nhiều câu lệnh trong khối thì phải sử dụng cặp dấu ngoặc nhọn { }.

Điều kiện cho câu lệnh if có thể bao gồm nhiều biểu thức. Câu lệnh if chỉ thực hiện hành động nếu kết quả cuối cùng của biểu thức là **true**(đúng).

Ví dụ

|  |
| --- |
| if ((a > 1) && (b > 1)) {  puts("a lớn hơn 1");  a++;  } |

sẽ chỉ thực hiện printf và a++ nếu cả a và b đều lớn hơn 1.

## PHẦN 16.2 NESTED IF()… ELSE VS IF ELSE LADDER ( CÂU LỆNH IF LỒNG NHAU VÀ CÂU LỆNH IF BẬC THANG)

Các câu lệnh **if()...else lồng nhau** tốn nhiều thời gian thực thi hơn (chúng chạy chậm hơn) so với một chuỗi **if()...else**, bởi vì các câu lệnh **if()...else** lồng nhau kiểm tra tất cả các điều kiện bên trong sau khi câu lệnh **if()** bên ngoài được thỏa mãn, trong khi chuỗi **if()..else** dừng kiểm tra điều kiện ngay khi bất kỳ điều kiện **if()** hoặc **else if()** nào được thỏa mãn.

**if()… else bậc thang**

|  |
| --- |
| #include<stdio.h>  int main(int *argc*, char \**argv*[])  {  int a, b, c;  printf("\nEnter Three numbers = ");  scanf("%d%d%d", &a, &b, &c);  if ((a < b) && (a < c))  {  printf("\na = %d is the smallest.", a);  }  else if ((b < a) && (b < c))  {  printf("\nb = %d is the smallest.", b);  }  else if ((c < a) && (c < b))  {  printf("\nc = %d is the smallest.", c);  }  else  {  printf("\nImprove your coding logic");  }  return 0;  } |

**if()… else lồng nhau**

|  |
| --- |
| #include <stdio.h>  int main(int *argc*, char \**argv*[])  {  int a, b, c;  printf("\nEnter Three numbers = ");  scanf("%d%d%d", &a, &b, &c);  if (a < b)  {  if (a < c)  {  printf("\na = %d is the smallest.", a);  }  else  {  printf("\nc = %d is the smallest.", c);  }  }  else  {  if(b < c)  {  printf("\nb = %d is the smallest.", b);  }  else  {  printf("\nc = %d is the smallest.", c);  }  }  return 0;  } |

## PHẦN 16.3 CÂU LỆNH SWITCH()

Câu lệnh **switch** được sử dụng khi bạn muốn chương trình thực hiện nhiều hành động khác nhau tùy theo giá trị của một biến kiểm tra cụ thể.

Ví dụ sử dụng câu lệnh **switch** như sau:

|  |
| --- |
| int a = 1;  switch (a) {  case 1:  puts("a is 1");  break;  case 2:  puts("a is 2");  break;  default:  puts("a is neither 1 nor 2");  break;  } |

Ví dụ này tương đương với việc sử dụng câu lệnh if()...else if()...else:

|  |
| --- |
| int a = 1;  if (a == 1) {  puts("a is 1");  } else if (a == 2) {  puts("a is 2");  } else {  puts("a is neither 1 nor 2");  } |

Nếu giá trị của a là 1 khi sử dụng câu lệnh switch, sẽ in ra "a is 1". Nếu giá trị của a là 2, sẽ in ra "a is 2". Nếu không phù hợp với bất kỳ lựa chọn nào của case, sẽ in ra "a is neither 1 nor 2".

Trong câu lệnh switch, các trường hợp **case n**: được sử dụng để mô tả vị trí mà luồng thực thi sẽ nhảy vào khi giá trị được truyền vào câu lệnh switch là n. Giá trị của n phải là hằng số tại thời điểm biên dịch và n có thể tồn tại nhiều nhất một lần trong một câu lệnh switch.

Trường hợp **default**: được sử dụng để mô tả khi giá trị không phù hợp với bất kỳ lựa chọn nào của case. Một thói quen tốt là luôn bao gồm trường hợp default trong mỗi câu lệnh switch để xử lý hành vi không mong muốn.

Một câu lệnh **break**; được yêu cầu để thoát khỏi khối **switch**.

**Lưu ý:** Nếu bạn vô tình quên thêm **break** sau mỗi trường hợp **case**, trình biên dịch sẽ giả định rằng bạn có ý định thực hiện "rơi vào" (**fall through**) và tất cả các trường hợp case tiếp theo, nếu có, sẽ được thực thi (trừ khi có break trong bất kỳ trường hợp case nào sau đó). Điều này có thể được sử dụng để triển khai Duff's Device. Tính năng này thường được coi là một khuyết điểm trong quy định ngôn ngữ C.

Dưới đây là một ví dụ cho thấy hiệu ứng khi không có **break;:**

|  |
| --- |
| int a = 1;  switch (a) {  case 1:  case 2:  puts("a is 1 or 2");  case 3:  puts("a is 1, 2 or 3");  break;  default:  puts("a is neither 1, 2 nor 3");  break;  } |

Khi giá trị của a là 1 hoặc 2, cả "a is 1 or 2" và "a is 1, 2 or 3" đều được in ra. Khi a là 3, chỉ có "a is 1, 2 or 3" được in ra. Nếu không phù hợp với bất kỳ lựa chọn nào của case, sẽ in ra "a is neither 1, 2 nor 3".

Lưu ý rằng *case* **default** không nhất thiết phải có, đặc biệt khi tập hợp các giá trị bạn nhận được trong câu lệnh switch đã được hoàn thành và biết trước tại thời điểm biên dịch.

Ví dụ tốt nhất là sử dụng **switch** trên một **enum**.

|  |
| --- |
| enum msg\_type { ACK, PING, ERROR };  void f(enum *msg\_type* *t*)  {  switch (*t*) {  case ACK:  // không làm gì cả  break;  case PING:  // thực hiện một số công việc  break;  case ERROR:  // thực hiện công việc khác  break;  }  } |

Có nhiều lợi ích khi làm như vậy:

* Hầu hết các trình biên dịch sẽ báo lỗi nếu bạn không xử lý một giá trị (điều này sẽ không được báo lỗi nếu có trường hợp default).
* Vì cùng một lý do, nếu bạn thêm một giá trị mới vào enum, bạn sẽ nhận được thông báo về tất cả các vị trí mà bạn đã quên xử lý giá trị mới (nếu có trường hợp default, bạn sẽ phải tìm thủ công trong mã của bạn để tìm các trường hợp đó).
* Người đọc không cần phải xem xét "được ẩn bởi trường hợp default:", liệu có các giá trị enum khác hay không hoặc liệu nó là một sự bảo vệ "just in case". Nếu có các giá trị enum khác, liệu người viết mã có ý dùng trường hợp default cho chúng hay không, hay có lỗi xuất hiện khi người viết thêm giá trị?
* Xử lý từng giá trị enum khiến mã trở nên dễ hiểu vì bạn không thể che giấu sau một ký tự đại diện (wild card), bạn phải xử lý từng giá trị cụ thể.

Tuy nhiên, bạn không thể ngăn người khác viết mã xấu như sau:

|  |
| --- |
| enum *msg\_type* t = (enum *msg\_type*)666; // Tôi là evil |

Do đó, nếu bạn thực sự cần, bạn có thể thêm một kiểm tra bổ sung trước câu lệnh switch để phát hiện điều này.

|  |
| --- |
| void f(enum *msg\_type* *t*)  {  if (!is\_msg\_type\_valid(*t*)) {  // Xử lý lỗi không thường xảy ra này  }  switch(*t*) {  // Cùng mã như trước đó  }  } |

## PHẦN 16.4 CÂU LỆNH IF()… ELSE VÀ CÚ PHÁP

Trong khi lệnh if thực hiện một hành động chỉ khi điều kiện của nó **đúng**, if / else cho phép bạn xác định các hành động khác nhau khi điều kiện **đúng** và khi điều kiện **sai.**

Ví dụ:

|  |
| --- |
| if (a > 1)  puts("a is larger than 1");  else  puts("a is not larger than 1"); |

Tương tự như câu lệnh if, khi khối lệnh trong if hoặc else chỉ gồm một câu lệnh, dấu ngoặc nhọn có thể được bỏ qua (nhưng không nên làm điều này vì có thể dễ dàng gây ra các vấn đề không cần thiết). Tuy nhiên, nếu có nhiều hơn một câu lệnh trong khối lệnh if hoặc else, thì phải sử dụng dấu ngoặc nhọn cho khối đó.

|  |
| --- |
| if (a > 1)  {  puts("a is larger than 1");  a--;  }  else  {  puts("a is not larger than 1");  a++;  } |

## PHẦN 16.5 NỐI HAI HOẶC NHIỀU LỆNH IF()… ELSE

Trong khi lệnh if () ... else chỉ cho phép xác định một (mặc định) hành vi xảy ra khi điều kiện trong lệnh if không được đáp ứng, việc nối hai hoặc nhiều lệnh if () ... else cho phép xác định một số hành vi khác trước khi điều kiện chuyển đến nhánh else cuối cùng, hành vi này được xem như "default", nếu có.

Ví dụ:

|  |
| --- |
| int a = ... /\* khởi tạo với một giá trị nào đó. \*/  if (a >= 1)  {  printf("a is greater than or equals 1.\n");  }  else if (a == 0) // ta đã biết rằng a nhỏ hơn 1  {  printf("a equals 0.\n");  }  else /\* a nhỏ hơn 1 và không bằng 0, do đó: \*/  {  printf("a is negative.\n");  } |

# INITIALIZATION (KHỞI TẠO)

## PHẦN 17.1 KHỞI TẠO BIẾN TRONG C

Trong trường hợp không có khởi tạo rõ ràng, các biến bên ngoài (biến toàn cục) và biến tĩnh(static) được đảm bảo được khởi tạo thành giá trị **không**; các biến tự động (bao gồm cả biến đăng ký) có giá trị ban đầu không xác định (tức là giá trị rác).

Các biến nguyên thủy (scalar) có thể được khởi tạo khi được định nghĩa bằng cách đặt dấu bằng và biểu thức sau tên biến:

|  |
| --- |
| int x = 1;  char squota = '\'';  long day = 1000L \* 60L \* 60L \* 24L; /\* milliseconds/day \*/ |

Đối với biến bên ngoài và tĩnh(static) bộ khởi tạo phải là một biểu thức hằng số; việc khởi tạo được thực hiện một lần, trước khi chương trình bắt đầu thực thi.

Đối với biến tự động và biến đăng ký, bộ khởi tạo không bị giới hạn chỉ là hằng số: nó có thể là bất kỳ biểu thức nào liên quan đến các giá trị đã được xác định trước, kể cả các lời gọi hàm.

Ví dụ xem đoạn mã dưới đây

|  |
| --- |
| int binsearch(int *x*, int *v*[], int *n*)  {  int low = 0;  int high = *n* - 1;  int mid;  ...  } |

Thay vì

|  |
| --- |
| int low, high, mid;  low = 0;  high = n - 1; |

Trong thực tế, việc khởi tạo các biến tự động chỉ là cách viết ngắn gọn cho các câu lệnh gán. Loại nào nên được ưu tiên phụ thuộc phần lớn vào sở thích. Chúng ta thường sử dụng các phép gán rõ ràng, vì việc khởi tạo trong các khai báo khó nhìn thấy hơn và xa so với điểm sử dụng. Tuy nhiên, biến chỉ nên được khai báo khi chúng được sử dụng bất cứ khi nào có thể.

### 17.1.1 Khởi tạo một mảng

Một mảng có thể được khởi tạo bằng cách theo sau khai báo của nó bằng một danh sách các giá trị khởi tạo nằm trong dấu ngoặc nhọn và cách nhau bằng dấu phẩy.

Ví dụ, để khởi tạo một mảng days với số ngày trong mỗi tháng:

|  |
| --- |
| int days\_of\_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; |

(Lưu ý rằng tháng 1 được mã hóa là tháng số 0 trong cấu trúc này.)

Khi kích thước của mảng bị bỏ qua, trình biên dịch sẽ tính độ dài bằng cách đếm các giá trị khởi tạo, trong trường hợp này có 12 giá trị.

Nếu có ít giá trị khởi tạo hơn số lượng chỉ định, những giá trị còn lại sẽ là giá trị không(0) cho tất cả các loại biến. Nếu có quá nhiều bộ khởi tạo, đó là lỗi. Không có cách tiêu chuẩn nào để chỉ định lặp lại bộ khởi tạo - nhưng GCC có một phần mở rộng để làm điều đó.

Phiên bản < C99

Trong C89/C90 hoặc các phiên bản C trước đó, không có cách nào để khởi tạo một phần tử ở giữa một mảng mà không cung cấp tất cả các giá trị trước đó.

Phiên bản ≥ C99

Với C99 và các phiên bản sau đó, các khởi tạo đặc chỉ định cho phép bạn khởi tạo các phần tử tùy ý của một mảng, để lại các giá trị chưa được khởi tạo như số 0.

### 17.1.2 Khởi tạo mảng kí tự

Trong ngôn ngữ lập trình C, khởi tạo mảng ký tự có một cách viết ngắn gọn đặc biệt. Chúng ta có thể sử dụng một chuỗi (string) để khởi tạo mảng ký tự thay vì sử dụng cặp dấu ngoặc nhọn và dấu phẩy.

Ví dụ:

|  |
| --- |
| char chr\_array[] = "hello"; |

Đoạn mã trên là một cách viết ngắn gọn tương đương với:

|  |
| --- |
| char chr\_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' }; |

Trong trường hợp này, mảng chr\_array có kích thước là 6 (gồm 5 ký tự và ký tự null '\0' để đánh dấu kết thúc chuỗi).

1. Biến không được khởi tạo (uninitialized variable) trong C sẽ có giá trị không xác định (rác). Giá trị của biến không được đảm bảo và có thể chứa các giá trị ngẫu nhiên từ bộ nhớ trước đó. Việc sử dụng biến không được khởi tạo có thể dẫn đến hành vi không xác định và kết quả không đáng tin cậy trong chương trình.
2. Một biểu thức hằng số (constant expression) trong C là một biểu thức có thể được đánh giá ngay tại thời điểm biên dịch (compile-time). Một biến có từ khóa "const" chỉ là biến chỉ đọc (read-only) và không phải là một biểu thức hằng số. Do đó, các biến có từ khóa "const" không thể được sử dụng để xác định kích thước của mảng trong C. Ví dụ sau không hợp lệ:

|  |
| --- |
| const int SIZE = 10;  int global\_arr[SIZE]; |

Và cũng không thể sử dụng biến "const" để khởi tạo mảng, như sau:

|  |
| --- |
| const int SIZE = 10;  int global\_var = SIZE; |

Điều này không hợp lệ trong C vì kích thước của mảng phải là một hằng số đã biết tại thời điểm biên dịch, không thể là một biến có từ khóa "const" có giá trị không biết trước.

## PHẦN 17.2 SỬ DỤNG DESIGNATED INITIALIZERS

Trong phiên bản C99 trở lên, ngôn ngữ C hỗ trợ (designated initializers). Điều này cho phép bạn chỉ định các phần tử của một array, structure hoặc union sẽ được khởi tạo bằng các giá trị theo sau.

### 17.2.1 Designated initializers cho các phần tử mảng

Với kiểu dữ liệu đơn giản như int:

|  |
| --- |
| int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 }; |

Phần tử nằm trong dấu ngoặc vuông và là một biểu thức nguyên hằng số, xác định phần tử của mảng sẽ được khởi tạo bằng giá trị của biểu thức sau dấu bằng. Các phần tử không được chỉ định sẽ được khởi tạo mặc định là 0. Ví dụ này chỉ ra việc “designated initializers” theo thứ tự; không cần phải theo thứ tự. Ví dụ cho thấy có một số khoảng trống; điều đó là hợp lệ. Ví dụ không hiển thị hai khởi tạo khác nhau cho cùng một phần tử; điều này cũng được phép.

Trong ví dụ này, kích thước của mảng không được xác định rõ ràng, do đó chỉ số tối đa được chỉ định trong các khởi tạo dùng định danh xác định kích thước của mảng - trong ví dụ này là 21 phần tử. Nếu kích thước được xác định rõ ràng, việc khởi tạo một phần tử vượt quá kích thước của mảng sẽ là lỗi, như thường lệ.

### 17.2.2 Designated initializers cho structures

Bạn có thể chỉ định các thành phần của một cấu trúc được khởi tạo bằng cách sử dụng cú pháp ***.element***:

|  |
| --- |
| struct *Date*  {  int year;  int month;  int day;  };  struct *Date* us\_independence\_day = { .day = 4, .month = 7, .year = 1776 }; |

Nếu các thành phần không được liệt kê, chúng sẽ được khởi tạo mặc định (bằng 0).

### 17.2.3 Designated initializer cho unions

Bạn có thể chỉ định phần tử nào của union được khởi tạo bằng Designated initializer.

Trước tiêu chuẩn C, không có cách nào để khởi tạo một union. Tiêu chuẩn C89/C90 cho phép bạn khởi tạo thành viên đầu tiên của union — vì vậy việc lựa chọn thành viên nào được liệt kê đầu tiên là vấn đề quan trọng.

|  |
| --- |
| struct *discriminated\_union*  {  enum { DU\_INT, DU\_DOUBLE } discriminant;  union  {  int du\_int;  double du\_double;  } du;  };  struct *discriminated\_union* du1 = { .discriminant = DU\_INT, .du = { .du\_int = 1 } };  struct *discriminated\_union* du2 = { .discriminant = DU\_DOUBLE, .du = { .du\_double = 3.14159 } }; |

Lưu ý rằng C11 cho phép bạn sử dụng các thành viên union ẩn danh bên trong một structure, do đó bạn không cần tên du trong ví dụ trước:

|  |
| --- |
| struct *discriminated\_union*  {  enum { DU\_INT, DU\_DOUBLE } discriminant;  union  {  int du\_int;  double du\_double;  };  };  struct *discriminated\_union* du1 = { .discriminant = DU\_INT, .du\_int = 1 };  struct *discriminated\_union* du2 = { .discriminant = DU\_DOUBLE, .du\_double = 3.14159 }; |

### 17.2.4 Designated initializers cho arrays của structures, etc

Các cấu trúc này có thể được kết hợp cho các mảng cấu trúc chứa các phần tử là mảng, v.v. Việc sử dụng các bộ dấu ngoặc nhọn đầy đủ đảm bảo rằng ký hiệu là rõ ràng.

|  |
| --- |
| typedef struct *Date* {  int year;  int month;  int day;  } *Date*;  struct *date\_range* {  *Date* dr\_from;  *Date* dr\_to;  char dr\_what[80];  };  struct *date\_range* ranges[] = {  [3] = { .dr\_from = { .year = 1066, .month = 10, .day = 14 },  .dr\_to = { .year = 1066, .month = 12, .day = 25 },  .dr\_what = "Chiến tranh Hastings đến Lễ đăng quang của William the Conqueror"  },  [2] = { .dr\_from = { .month = 7, .day = 4, .year = 1776 },  .dr\_to = { .month = 5, .day = 14, .year = 1787 },  .dr\_what = "Tuyên ngôn Độc lập Mỹ đến Hội nghị Hiến pháp"  }  }; |

### 17.2.5 Specifying ranges in array initializers (Chỉ định phạm vi trong bộ khởi tạo mảng)

GCC cung cấp một phần mở rộng cho phép bạn chỉ định một dải các phần tử trong mảng sẽ được khởi tạo bằng cùng một giá trị. Điều này giúp viết mã nguồn ngắn gọn hơn khi bạn muốn khởi tạo một số phần tử liên tiếp trong mảng.

|  |
| --- |
| int array[] = { [3 ... 7] = 29, [19] = 107 }; |

các dấu ba chấm (...) cần phải được tách rời với các số bằng khoảng trống hoặc các ký tự khác, nếu không chúng có thể bị hiểu là một phần của một số thực (floating-point number) do "quy tắc ăn nhiều nhất" (maximal munch rule).

## PHẦN 17.3 INITIALIZING STRUCTURES AND ARRAYS OF STRUCTURES (KHỞI TẠO CẤU TRÚC VÀ MẢNG CẤU TRÚC)

Cấu trúc và mảng cấu trúc có thể được khởi tạo bởi một loạt các giá trị được đặt trong dấu ngoặc nhọn, một giá trị cho mỗi thành viên của cấu trúc.

|  |
| --- |
| struct *Date*  {  int year;  int month;  int day;  };  struct *Date* us\_independence\_day = { 1776, 7, 4 };  struct *Date* uk\_battles[] =  {  { 1066, 10, 14 }, // Battle of Hastings  { 1815, 6, 18 }, // Battle of Waterloo  { 1805, 10, 21 }, // Battle of Trafalgar  }; |

Chú ý rằng việc khởi tạo mảng có thể được viết mà không cần dấu ngoặc nhọn bên trong, và trong quá khứ (trước năm 1990, chẳng hạn) thường sẽ được viết mà không có chúng:

|  |
| --- |
| struct *Date* uk\_battles[] =  {  1066, 10, 14, // Battle of Hastings  1815, 6, 18, // Battle of Waterloo  1805, 10, 21, // Battle of Trafalgar  }; |

Mặc dù cách viết này vẫn hoạt động, nhưng không phải là phong cách hiện đại tốt - bạn không nên cố gắng sử dụng cú pháp này trong mã mới và nên sửa các cảnh báo của trình biên dịch nó thường thông báo.

Hãy xem thêm về designated initializers.

# KHAI BÁO VÀ ĐỊNH NGHĨA

## Hiểu về khai báo và định nghĩa

Một khai báo giới thiệu một định danh và mô tả kiểu dữ liệu của nó, cho dù là kiểu, đối tượng hoặc hàm. Một khai báo là điều mà trình biên dịch cần để chấp nhận các tham chiếu đến định danh đó. Dưới đây là các khai báo:

|  |
| --- |
| extern int bar;  extern int g(int, int);  double f(int, double); /\* từ khóa extern có thể bỏ qua cho khai báo hàm \*/  double h1(); /\* khai báo hàm không có nguyên mẫu \*/  double h2(); /\* tương tự cho khai báo hàm \*/ |

Một định nghĩa thực sự khởi tạo/hiện thực hóa định danh này. Đây là điều mà trình liên kết cần để liên kết các tham chiếu đến các thực thể đó. Dưới đây là các định nghĩa tương ứng với các khai báo ở trên:

|  |
| --- |
| int bar;  int g(int *lhs*, int *rhs*) { return *lhs* \* *rhs*; }  double f(int *i*, double *d*) { return *i* + *d*; }  double h1(int *a*, int *b*) { return -1.5; }  double h2() {} /\* nguyên mẫu được ngụ ý trong định nghĩa, tương tự double h2(void) \*/ |

Một định nghĩa có thể được sử dụng thay thế cho một khai báo.

Tuy nhiên, nó phải được định nghĩa đúng một lần. Nếu bạn quên định nghĩa một tên mà đã được khai báo và được tham chiếu ở nơi khác, trình liên kết sẽ không biết liên kết các tham chiếu đó với gì và báo lỗi về các ký hiệu bị thiếu. Ngược lại, nếu bạn định nghĩa một tên nhiều hơn một lần, trình liên kết không biết liên kết các tham chiếu với phiên bản nào và báo lỗi về các ký hiệu bị trùng lặp.

**Ngoại lệ:**

|  |
| --- |
| extern int i = 0; /\* định nghĩa 'i' với giá trị khởi tạo là 0 \*/  extern int j; /\* chỉ khai báo 'j', không định nghĩa \*/ |

Trường hợp này có thể giải thích bằng các khái niệm "Strong symbols vs Weak symbols" (từ góc nhìn của trình liên kết). Vui lòng xem tại đây (Slide 22) để biết thêm giải thích.

|  |
| --- |
| /\* All are definitions. \*/  struct *S* { int a; int b; }; /\* defines S \*/  struct *X* { /\* defines X \*/  int x; /\* defines non-static data member x \*/  };  struct *X* anX; /\* defines anX \*/ |

# COMMAND- LINE ARGUMENTS (ĐỐI SỐ DÒNG LỆNH)

|  |  |
| --- | --- |
| **Tham số** | **Chi tiết** |
| argc | argument count(số lượng đối số) - được khởi tạo thành số lượng đối số được phân tách bằng dấu cách được cung cấp cho chương trình từ dòng lệnh cũng như chính tên chương trình. |
| argv | argument vector (vectơ đối số) - được khởi tạo thành một mảng các con trỏ ký tự (chuỗi) chứa các đối số (và tên chương trình) đã được đưa ra trên dòng lệnh. |

## In ra các đối số của chương trình và chuyển đổi thành các giá trị số nguyên

Đoạn mã dưới đây sẽ in ra các đối số của chương trình và cố gắng chuyển đổi mỗi đối số thành một số (kiểu long):

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <errno.h>  #include <limits.h>  int main(int *argc*, char\* *argv*[]) {  for (int i = 1; i < *argc*; i++) {  printf("Tham so %d la: %s\n", i, *argv*[i]);  errno = 0;  char \*p;  long argument\_numValue = strtol(*argv*[i], &p, 10);  if (p == *argv*[i]) {  fprintf(stderr, "Tham so %d khong phai la mot so.\n", i);  }  else if ((argument\_numValue == LONG\_MIN || argument\_numValue == LONG\_MAX) && errno == ERANGE) {  fprintf(stderr, "Tham so %d vuot qua pham vi.\n", i);  }  else {  printf("Tham so %d la mot so, va gia tri la: %ld\n", i, argument\_numValue);  }  }  return 0;  } |

Tham khảo:

* strtol() trả về giá trị không chính xác
* Cách sử dụng chính xác của strtol

## PHẦN 19.2 IN RA CÁC ĐỐI SỐ CỦA DÒNG LỆNH

Sau khi nhận được các đối số, bạn có thể in chúng như sau:

|  |
| --- |
| int main(int *argc*, char \*\**argv*)  {  for (int i = 1; i < *argc*; i++)  {  printf("Doi so %d: [%s]\n", i, *argv*[i]);  }  } |

**Ghi chú:**

1. Tham số argv có thể được định nghĩa là char \*argv[].
2. argv[0] có thể chứa tên chương trình chính (phụ thuộc vào cách chương trình được thực thi). Tham số dòng lệnh đầu tiên là argv[1], và đây là lý do tại sao biến vòng lặp i được khởi tạo bằng 1.
3. Trong câu lệnh in, bạn có thể sử dụng \*(argv + i) thay vì argv[i] - nó trả về cùng một giá trị, nhưng dài dòng hơn.
4. Dấu ngoặc vuông xung quanh giá trị tham số giúp xác định điểm bắt đầu và kết thúc. Điều này rất hữu ích nếu có các dấu cách cuối, ký tự mới, ký tự nằm ngang, hoặc các ký tự lạ trong tham số. Một biến thể của chương trình này là một công cụ hữu ích để gỡ lỗi các shell script khi bạn cần hiểu thực sự tham số dòng lệnh chứa gì (mặc dù có một số lựa chọn đơn giản trong shell gần như tương tự).

## PHẦN 19.3 SỬ DỤNG CÔNG CỤ GETOPT CỦA GNU

Các tùy chọn dòng lệnh cho các ứng dụng không được ngôn ngữ C xử lý khác với các đối số dòng lệnh. Chúng chỉ là các đối số, trong môi trường Linux hoặc Unix, theo truyền thống bắt đầu bằng dấu gạch ngang (-).

Với glibc trong môi trường Linux hoặc Unix, bạn có thể sử dụng các công cụ getopt để dễ dàng xác định, xác thực và phân tích các **command line** từ phần còn lại của các đối số của bạn.

Các công cụ này yêu cầu các tùy chọn của bạn được định dạng theo Tiêu chuẩn mã hóa GNU, đây là phần mở rộng của những gì POSIX chỉ định cho định dạng của các **command line**.

Ví dụ dưới đây minh họa việc xử lý các tùy **command line** với các công cụ getopt của GNU.

|  |
| --- |
|  |

Nó có thể được biên dịch với gcc:

|  |
| --- |
| gcc example.c -o example |

Chương trình này hỗ trợ ba tùy chọn dòng lệnh (--help, --file và --msg). Tất cả đều có một "dạng rút gọn" (-h, -f và -m).

Tùy chọn "file" và "msg" đều có thể nhận đối số. Nếu bạn chỉ định tùy chọn "msg", đối số của nó là bắt buộc.

Các đối số cho các tùy chọn được định dạng như sau:

* --option=value (cho các tùy chọn dạng dài)
* -o value hoặc -o "value" (cho các tùy chọn dạng ngắn)

# FILES VÀ LUỒNG INPUT/OUTPUT

|  |  |
| --- | --- |
| **Tham số** | **Chi tiết** |
| const char \*mode | Một chuỗi mô tả chế độ mở của luồng tệp tin. Xem phần ghi chú để biết các giá trị có thể. |
| int whence | Có thể là SEEK\_SET để đặt vị trí từ đầu tệp tin, SEEK\_END để đặt vị trí từ cuối tệp tin, hoặc SEEK\_CUR để đặt vị trí liên quan đến giá trị con trỏ hiện tại. Lưu ý: SEEK\_END không có tính di động. |

## Mở và ghi vào tệp tin

Sử dụng hàm **fopen( )** để tạo một tệp mới hoặc mở một tệp hiện có. Cuộc gọi này sẽ khởi tạo một đối tượng thuộc loại **FILE** , chứa tất cả thông tin cần thiết để kiểm soát luồng. Nguyên mẫu của lệnh gọi hàm này như sau:

|  |
| --- |
| FILE \*fopen( const char \* filename, const char \* mode ); |

Ở đây, **tên tệp** là một chuỗi ký tự mà bạn sẽ sử dụng để đặt tên cho tệp của mình và **chế độ** truy cập có thể có một trong các giá trị sau:

|  |  |
| --- | --- |
| **STT** | **Mode và Mô tả** |
| 1 | **r**  Mở một tệp văn bản hiện có cho mục đích đọc. |
| 2 | **w**  Mở một tập tin văn bản để viết. Nếu nó không tồn tại thì một tập tin mới sẽ được tạo. Ở đây chương trình của bạn sẽ bắt đầu viết nội dung từ đầu file. |
| 3 | **a**  Mở một tệp văn bản để viết ở chế độ nối thêm. Nếu nó không tồn tại thì một tập tin mới sẽ được tạo. Tại đây chương trình của bạn sẽ bắt đầu nối thêm nội dung vào nội dung tệp hiện có. |
| 4 | **r+**  Mở một tập tin văn bản để đọc và viết. |
| 5 | **w+**  Mở một tập tin văn bản để đọc và viết. Đầu tiên, nó cắt ngắn tệp về độ dài bằng 0 nếu nó tồn tại, mặt khác sẽ tạo một tệp nếu nó không tồn tại. |
| 6 | **a+**  Mở một tập tin văn bản để đọc và viết. Nó tạo tập tin nếu nó không tồn tại. Việc đọc sẽ bắt đầu lại từ đầu nhưng việc viết chỉ có thể được nối thêm. |

Nếu bạn định xử lý các tệp nhị phân, thì bạn sẽ sử dụng các chế độ truy cập sau thay vì các chế độ được đề cập ở trên: "rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

|  |
| --- |
| #include <stdio.h> /\* for perror(), fopen(), fputs() and fclose() \*/  #include <stdlib.h> /\* for the EXIT\_\* macros \*/  int main(int *argc*, char \*\**argv*) {  int e = EXIT\_SUCCESS;  /\* Lấy đường dẫn từ tham số của hàm main, nếu không có thì mặc định là output.txt \*/  char \*path = (*argc* > 1) ? *argv*[1] : "output.txt";  /\* Mở tệp tin để ghi và nhận con trỏ tệp tin \*/  *FILE* \*file = fopen(path, "w");  /\* In thông báo lỗi và thoát nếu fopen() thất bại \*/  if (!file) {  perror(path);  return EXIT\_FAILURE;  }  /\* Ghi văn bản vào tệp tin. Không giống như puts(), fputs() không tự động thêm dòng mới. \*/  if (fputs("Nội dung xuất ra tệp tin.\n", file) == EOF)  {  perror(path);  e = EXIT\_FAILURE;  }  /\* Đóng tệp tin \*/  if (fclose(file))  {  perror(path);  return EXIT\_FAILURE;  }  return e;  } |

Chương trình này mở tệp tin có tên được đưa ra trong đối số của hàm main, nếu không có đối số nào được truyền thì mặc định là "*output.txt*". Nếu tệp tin đã tồn tại, nó sẽ bị xóa và được coi như một tệp mới trống. Nếu tệp tin chưa tồn tại, hàm **fopen()** sẽ tạo mới tệp tin đó.

Nếu cuộc gọi **fopen()** thất bại vì một lý do nào đó, nó sẽ trả về giá trị **NULL** và thiết lập giá trị biến toàn cục errno. Điều này cho phép chương trình kiểm tra giá trị được trả về sau cuộc gọi **fopen()** và sử dụng **perror()** để hiển thị thông báo lỗi tương ứng.

Nếu cuộc gọi **fopen()** thành công, nó sẽ trả về con trỏ FILE hợp lệ. Con trỏ này sau đó có thể được sử dụng để tham chiếu đến tệp tin này cho đến khi **fclose()** được gọi.

Hàm **fputs()** được sử dụng để ghi nội dung văn bản vào tệp tin đã mở, thay thế bất kỳ nội dung trước đó của tệp. Tương tự như **fopen(),** hàm **fputs()** cũng thiết lập giá trị errno nếu gặp lỗi, trong trường hợp này hàm sẽ trả về **EOF** để chỉ ra lỗi (nếu không xảy ra lỗi, hàm sẽ trả về một giá trị không âm).

Hàm **fclose()** được sử dụng để đóng tệp tin, đồng thời giải phóng bộ nhớ được trỏ bởi FILE \*. Giá trị trả về của hàm chỉ ra hoàn thành như hàm **fputs()** (tuy nhiên nó trả về '0' nếu thành công), đồng thời cũng thiết lập giá trị errno trong trường hợp xảy ra lỗi.

## Đóng một tập tin

Để đóng một tập tin, hãy sử dụng hàm fclose(). Nguyên mẫu của hàm này là:

|  |
| --- |
| int fclose( FILE \*fp ); |

Hàm fclose **(-)** trả về 0 nếu thành công hoặc **EOF** nếu có lỗi khi đóng tệp. Chức năng này thực sự xóa mọi dữ liệu vẫn đang chờ xử lý trong bộ đệm vào tệp, đóng tệp và giải phóng mọi bộ nhớ được sử dụng cho tệp. EOF là hằng số được xác định trong tệp tiêu đề **stdio.h** .

Có nhiều hàm khác nhau được thư viện chuẩn C cung cấp để đọc và ghi tệp, từng ký tự hoặc ở dạng chuỗi có độ dài cố định.

## RUN PROCESS (QUÁ TRÌNH CHẠY)

|  |
| --- |
| #include <stdio.h>  void print\_all(*FILE* \**stream*)  {  int c;  while ((c = getc(*stream*)) != EOF)  putchar(c);  }  int main(void)  {  *FILE* \*stream;  /\* gọi lệnh netstat. netstat có sẵn cho cả Windows và Linux \*/  if ((stream = popen("netstat", "r")) == NULL)  return 1;  print\_all(stream);  pclose(stream);  return 0;  } |

Chương trình này chạy một tiến trình (netstat) thông qua hàm popen() và đọc toàn bộ đầu ra chuẩn từ tiến trình đó và in ra đầu ra chuẩn của chính chương trình.

Lưu ý: Hàm popen() không tồn tại trong thư viện chuẩn C, mà thay vào đó nó là một phần của chuẩn POSIX C.

## PHẦN 20.3 FPRINTF

Bạn có thể sử dụng hàm fprintf để ghi dữ liệu vào tệp giống như bạn sử dụng printf để in ra màn hình. Ví dụ, để lưu trữ số lượng chiến thắng, thua và hòa trong trò chơi, bạn có thể viết như sau:

|  |
| --- |
| /\* Lưu trữ số lượng chiến thắng, hòa và thua vào tệp \*/  void savewlt(*FILE* \**fout*, int *wins*, int *losses*, int *ties*)  {  fprintf(*fout*, "Wins: %d\nTies: %d\nLosses: %d\n", *wins*, *ties*, *losses*);  } |

**Lưu ý**: Một số hệ thống (ví dụ: Windows) không sử dụng những gì mà hầu hết các lập trình viên gọi là "kết thúc dòng bình thường". Trong khi các hệ thống giống UNIX sử dụng \n để kết thúc dòng, Windows sử dụng cặp ký tự: \r (carriage return - CR) và \n (line feed - LF). Chuỗi này thường được gọi là CRLF. Tuy nhiên, khi sử dụng C, bạn không cần phải lo lắng về các chi tiết phụ thuộc vào nền tảng này. Trình biên dịch C sẽ tự động chuyển đổi mọi trường hợp của \n thành kết thúc dòng phù hợp với nền tảng. Vì vậy, trình biên dịch Windows sẽ chuyển đổi \n thành \r\n, trong khi trình biên dịch UNIX sẽ giữ nguyên \n.

## PHẦN 20.4 LẤY CÁC DÒNG TỪ TỆP SỬ DỤNG GETLINE()

Thư viện POSIX C định nghĩa hàm getline(). Hàm này cấp phát một bộ đệm để chứa nội dung của dòng và trả về dòng mới, số ký tự trong dòng và kích thước của bộ đệm.

Dưới đây là ví dụ chương trình lấy từng dòng của tệp *example.txt*

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #define FILENAME "example.txt"  int main(void)  {  /\* Mở tệp để đọc \*/  char \*line\_buf = NULL;  *size\_t* line\_buf\_size = 0;  int line\_count = 0;  ssize\_t line\_size;  *FILE* \*fp = fopen(FILENAME, "r");  if (!fp)  {  fprintf(stderr, "Error opening file '%s'\n", FILENAME);  return EXIT\_FAILURE;  }  /\* Lấy dòng đầu tiên của tệp. \*/  line\_size = getline(&line\_buf, &line\_buf\_size, fp);  /\* Lặp qua tệp cho đến khi kết thúc tệp. \*/  while (line\_size >= 0)  {  /\* Tăng số lượng dòng \*/  line\_count++;  /\* Hiển thị chi tiết của dòng \*/  printf("line[%06d]: chars=%06zd, buf size=%06zu, contents: %s", line\_count,  line\_size, line\_buf\_size, line\_buf);  /\* Lấy dòng tiếp theo \*/  line\_size = getline(&line\_buf, &line\_buf\_size, fp);  }  /\* Giải phóng bộ đệm dòng đã cấp phát \*/  free(line\_buf);  line\_buf = NULL;  /\* Đóng tệp khi đã hoàn thành \*/  fclose(fp);  return EXIT\_SUCCESS;  } |

**Input file example.txt**

|  |
| --- |
| This is a file  which has  multiple lines  with various indentation,  blank lines  a really long line to show that getline() will reallocate the line buffer if the length of a line  is too long to fit in the buffer it has been given,  and punctuation at the end of the lines. |

**Kết quả**

|  |
| --- |
| line[000001]: chars=000015, buf size=000016, contents: This is a file  line[000002]: chars=000012, buf size=000016, contents: which has  line[000003]: chars=000015, buf size=000016, contents: multiple lines  line[000004]: chars=000030, buf size=000032, contents: with various indentation,  line[000005]: chars=000012, buf size=000032, contents: blank lines  line[000006]: chars=000001, buf size=000032, contents:  line[000007]: chars=000001, buf size=000032, contents:  line[000008]: chars=000001, buf size=000032, contents:  line[000009]: chars=000150, buf size=000160, contents: a really long line to show that getline()  will reallocate the line buffer if the length of a line is too long to fit in the buffer it has  been given,  line[000010]: chars=000042, buf size=000160, contents: and punctuation at the end of the lines.  line[000011]: chars=000001, buf size=000160, contents: |

**Ví dụ khai triển getline()**

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <errno.h>  #include <stdint.h>  #if !(defined \_POSIX\_C\_SOURCE)  typedef long int *ssize\_t*;  #endif  /\* Chỉ bao gồm phiên bản getline() nếu phiên bản POSIX không khả dụng. \*/  #if !(defined \_POSIX\_C\_SOURCE) || \_POSIX\_C\_SOURCE < 200809L  #if !(defined SSIZE\_MAX)  #define SSIZE\_MAX (SIZE\_MAX >> 1)  #endif  /\* Hàm getline() thay thế cho phiên bản POSIX. \*/  *ssize\_t* getline(char \*\**pline\_buf*, *size\_t* \**pn*, *FILE* \**fin*)  {  const *size\_t* INITALLOC = 16;  const *size\_t* ALLOCSTEP = 16;  *size\_t* num\_read = 0;  // Kiểm tra xem các con trỏ đầu vào có NULL hay không.  if ((NULL == *pline\_buf*) || (NULL == *pn*) || (NULL == *fin*))  {  errno = EINVAL;  return -1;  }  // Nếu bộ đệm đầu ra là NULL, cấp phát bộ đệm mới.  if (NULL == \**pline\_buf*)  {  \**pline\_buf* = malloc(INITALLOC);  if (NULL == \**pline\_buf*)  {  // Không thể cấp phát bộ đệm.  return -1;  }  else  {  // Ghi nhận kích thước bộ đệm tại thời điểm này.  \**pn* = INITALLOC;  }  }  // Tiến hành đọc từng ký tự từ tệp cho đến khi gặp dấu xuống dòng hoặc kết thúc tệp.  {  int c;  while (EOF != (c = getc(*fin*)))  {  // Ghi nhận đã đọc một ký tự.  num\_read++;  // Cấp phát lại bộ đệm nếu cần thêm không gian.  if (num\_read >= \**pn*)  {  *size\_t* n\_realloc = \**pn* + ALLOCSTEP;  char \*tmp = realloc(\**pline\_buf*, n\_realloc + 1); // +1 cho ký tự null kết thúc chuỗi.  if (NULL != tmp)  {  // Sử dụng bộ đệm mới và ghi nhận kích thước bộ đệm mới.  \**pline\_buf* = tmp;  \**pn* = n\_realloc;  }  else  {  // Lỗi, cho phép người gọi giải phóng bộ đệm.  return -1;  }  // Kiểm tra tràn số.  if (SSIZE\_MAX < \**pn*)  {  errno = ERANGE;  return -1;  }  }  // Thêm ký tự vào bộ đệm.  (\**pline\_buf*)[num\_read - 1] = (char)c;  // Thoát khỏi vòng lặp nếu gặp dấu xuống dòng.  if (c == '\n')  {  break;  }  }  // Ghi nhận nếu đã đến cuối tệp.  if (EOF == c)  {  errno = 0;  return -1;  }  }  // Kết thúc chuỗi bằng ký tự null.  (\**pline\_buf*)[num\_read] = '\0';  return (*ssize\_t*)num\_read;  }  #endif |

## PHẦN 20.5 FSCANF()

Giả sử chúng ta có một tệp văn bản và chúng tôi muốn đọc tất cả các từ trong tệp đó để thực hiện một số yêu cầu.

**file.txt**

|  |
| --- |
| This is just  a test file  to be used by fscanf() |

Sau đây là hàm thực hiện chính:

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  void printAllWords(*FILE* \*);  int main(void)  {  *FILE* \*fp;  if ((fp = fopen("file.txt", "r")) == NULL) {  perror("Error opening file");  exit(EXIT\_FAILURE);  }  printAllWords(fp);  fclose(fp);  return EXIT\_SUCCESS;  }  void printAllWords(*FILE* \* *fp*)  {  char tmp[20];  int i = 1;  while (fscanf(*fp*, "%19s", tmp) != EOF) {  printf("Word %d: %s\n", i, tmp);  i++;  }  } |

**Kết quả:**

|  |
| --- |
| Word 1: This  Word 2: is  Word 3: just  Word 4: a  Word 5: test  Word 6: file  Word 7: to  Word 8: be  Word 9: used  Word 10: by  Word 11: fscanf() |

## PHẦN 20.6 ĐỌC CÁC DÒNG TỪ TẬP TIN

Tiêu đề "stdio.h" định nghĩa hàm fgets(). Hàm này đọc một dòng từ một luồng và lưu trữ nó vào một chuỗi đã xác định. Hàm này dừng việc đọc văn bản từ luồng khi một trong các điều kiện sau được thỏa mãn: đọc được n - 1 ký tự, đọc được ký tự xuống dòng ('\n'), hoặc đọc tới cuối tệp (EOF).

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MAX\_LINE\_LENGTH 80  int main(int *argc*, char \*\**argv*)  {  char \*path;  char line[MAX\_LINE\_LENGTH] = {0};  unsigned int line\_count = 0;  if (*argc* < 1)  return EXIT\_FAILURE;  path = *argv*[1];  /\* Open file \*/  *FILE* \*file = fopen(path, "r");  if (!file)  {  perror(path);  return EXIT\_FAILURE;  }  /\* Get each line until there are none left \*/  while (fgets(line, MAX\_LINE\_LENGTH, file))  {  /\* Print each line \*/  printf("line[%06d]: %s", ++line\_count, line);  /\* Add a trailing newline to lines that don't already have one \*/  if (line[strlen(line) - 1] != '\n')  printf("\n");  }  /\* Close file \*/  if (fclose(file))  {  return EXIT\_FAILURE;  perror(path);  }  } |

Gọi chương trình với đối số là đường dẫn đến tệp chứa văn bản sau:

|  |
| --- |
| This is a file  which has  multiple lines  with various indentation,  blank lines  a really long line to show that getline() will reallocate the  line buffer if the length of a line  is too long to fit in the buffer it has been given,  and punctuation at the end of the lines. |

Sẽ dẫn đến đầu ra sau:

|  |
| --- |
| line[000001]: This is a file  line[000002]: which has  line[000003]: multiple lines  line[000004]: with various indentation,  line[000005]: blank lines  line[000006]:  line[000007]:  line[000008]:  line[000009]:  line[000010]: a really long line to show that getline() will reallocate the line buffer if th  line[000011]: e length of a line  line[000012]: is too long to fit in the buffer it has been given,  line[000013]: and punctuation at the end of the lines. |

Ví dụ rất đơn giản này cho phép độ dài dòng tối đa cố định, sao cho các dòng dài hơn sẽ được tính là hai dòng một cách hiệu quả. Hàm fgets() yêu cầu mã cuộc gọi cung cấp bộ nhớ được sử dụng làm đích cho dòng được đọc.

POSIX làm cho chức năng getline() khả dụng, thay vào đó cấp phát bộ nhớ bên trong để mở rộng bộ đệm khi cần thiết cho một dòng có độ dài bất kỳ (miễn là có đủ bộ nhớ).

## MỞ VÀ VIẾT VÀO FILE NHỊ PHÂN

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void)  {  int result = EXIT\_SUCCESS;  char file\_name[] = "output.bin"; // Tên tệp nhị phân đầu ra  char str[] = "This is a binary file example"; // Chuỗi cần ghi vào tệp nhị phân  *FILE* \*fp = fopen(file\_name, "wb"); // Mở tệp nhị phân trong chế độ ghi nhị phân (write binary mode)  if (fp == NULL) // Nếu xảy ra lỗi khi tạo tệp  {  result = EXIT\_FAILURE;  fprintf(stderr, "fopen() failed for '%s'\n", file\_name); // In thông báo lỗi  }  else  {  *size\_t* element\_size = sizeof \*str; // Kích thước của một phần tử trong chuỗi str (1 byte)  *size\_t* elements\_to\_write = sizeof str; // Số lượng phần tử cần ghi (bao gồm ký tự kết thúc chuỗi '\0')    // Sử dụng hàm fwrite để ghi chuỗi str vào tệp  *size\_t* elements\_written = fwrite(str, element\_size, elements\_to\_write, fp);    if (elements\_written != elements\_to\_write) // Kiểm tra việc ghi dữ liệu vào tệp  {  result = EXIT\_FAILURE;  fprintf(stderr, "fwrite() failed: wrote only %zu out of %zu elements.\n",  elements\_written, elements\_to\_write); // In thông báo lỗi  }  fclose(fp); // Đóng tệp sau khi ghi xong  }  return result;  } |

Chương trình này tạo và ghi dữ liệu dưới dạng nhị phân vào tệp output.bin bằng cách sử dụng hàm fwrite.

Nếu tệp đã tồn tại trước đó, nội dung của tệp sẽ bị xóa và tệp sẽ được xem như là một tệp mới trống.

Một luồng nhị phân là một chuỗi ký tự được sắp xếp có thể ghi lại dữ liệu nội tại một cách (transparently). Trong chế độ này, các byte được ghi giữa chương trình và tệp mà không có bất kỳ chuyển đổi nào.

Để ghi các số nguyên một cách di động, cần phải biết định dạng tệp mong đợi chúng ở dạng lưu trữ big-endian hay little-endian, và kích thước của chúng (thường là 16, 32 hoặc 64 bit). Bit shifting và masking có thể được sử dụng để ghi các byte theo thứ tự chính xác. Tuy nhiên, các số nguyên trong C không được đảm bảo có biểu diễn bù hai (two's complement) (mặc dù hầu hết các bản cài đặt đều sử dụng biểu diễn này). May mắn thay, chuyển đổi sang số không dấu được đảm bảo sử dụng biểu diễn bù hai. Do đó, mã nguồn để ghi số nguyên có dấu vào tệp nhị phân là một chút bất ngờ.

|  |
| --- |
| int fput16le(int *x*, *FILE* \**fp*)  {  unsigned int rep = *x*;  int e1, e2;  e1 = fputc(rep & 0xFF, *fp*);  e2 = fputc((rep >> 8) & 0xFF, *fp*);    if(e1 == EOF || e2 == EOF)  return EOF;  return 0;  } |

Các chức năng khác theo cùng một mẫu với các sửa đổi nhỏ về kích thước và thứ tự byte

# FORMATTED INPUT/OUTPUT (ĐỊNH DẠNG ĐẦU VÀO ĐẦU RA)

## PHẦN 21.1 CÁC BỘ CHUYỂN ĐỔI CHO VIỆC IN RA

|  |  |  |
| --- | --- | --- |
| **Bộ chuyển đổi** | **Loại đối số** | **Mô tả** |
| i, d | int | In ra số nguyên dạng thập phân. |
| u | unsigned int | In ra số nguyên không âm dạng thập phân. |
| o | unsigned int | In ra số nguyên không âm dạng bát phân (octal). |
| x | unsigned int | In ra số nguyên không âm dạng thập lục phân (hexadecimal) với chữ thường. |
| X | unsigned int | In ra số nguyên không âm dạng thập lục phân (hexadecimal) với chữ in hoa. |
| f | double | In ra số thực (float) với độ chính xác mặc định là 6 chữ số thập phân, nếu không xác định độ chính xác (dùng chữ thường cho các số đặc biệt như nan và inf). |
| F | double | Tương tự như bộ chuyển đổi f, nhưng dùng chữ in hoa cho các số đặc biệt như NAN và INF. |
| e | double | In ra số thực (float) với độ chính xác mặc định là 6 chữ số thập phân, nếu không xác định độ chính xác, dùng cách viết dạng số học (mantissa/exponent) với chữ thường cho số đặc biệt. |
| E | double | Tương tự như bộ chuyển đổi e, nhưng dùng chữ in hoa cho số đặc biệt. |
| g | double | Sử dụng bộ chuyển đổi f hoặc e [xem bên dưới]. |
| G | double | Sử dụng bộ chuyển đổi F hoặc E [xem bên dưới]. |
| a | double | In ra số thực (float) dưới dạng thập lục phân (hexadecimal) với chữ thường. |
| A | double | In ra số thực (float) dưới dạng thập lục phân (hexadecimal) với chữ in hoa. |
| c | char | In ra một ký tự duy nhất. |
| s | char\* | In ra chuỗi ký tự cho đến khi gặp ký tự null (NUL terminator), hoặc cắt chuỗi dựa trên độ chính xác (precision) nếu có. |
| p | void\* | In ra giá trị con trỏ không kiểu (void-pointer); con trỏ không kiểu khác nên được chuyển đổi ("cast") thành void\*; chỉ áp dụng cho con trỏ đối tượng, không phải con trỏ hàm. |
| % | n/a | In ra ký tự '%'. |
| n | int\* | Ghi số byte đã in ra cho đến thời điểm hiện tại vào địa chỉ được trỏ tới bởi con trỏ int\*. |

Lưu ý rằng các bộ chuyển đổi chiều dài có thể được áp dụng cho %n (ví dụ: %hhn chỉ ra rằng bộ chuyển đổi %n tiếp theo áp dụng cho một con trỏ trỏ tới đối số kiểu signed char, theo tiêu chuẩn ISO/IEC 9899:2011 §7.21.6.1 ¶7).

Lưu ý rằng các bộ chuyển đổi số thực áp dụng cho các kiểu float và double do quy tắc tăng cường mặc định - §6.5.2.2 Function calls, ¶7. Dấu ba chấm trong khai báo nguyên mẫu của hàm khiến việc chuyển đổi kiểu đối số dừng lại sau tham số cuối cùng. Các chuyển đổi mặc định của đối số được thực hiện trên các đối số cuối cùng.) Do đó, các hàm như printf() chỉ nhận giá trị double, ngay cả khi biến được tham chiếu là kiểu float.

Với các định dạng g và G, việc chọn giữa kiểu chữ số e và f (hoặc E và F) được mô tả trong tiêu chuẩn C và quy tắc POSIX cho printf():

* Đối số double đại diện cho một số dấu phẩy động sẽ được chuyển đổi theo kiểu f hoặc e (hoặc theo kiểu F hoặc E trong trường hợp của chuyển đổi G), tùy thuộc vào giá trị được chuyển đổi và độ chính xác.
* Cho P là độ chính xác nếu không bằng không, 6 nếu độ chính xác bị bỏ qua, hoặc 1 nếu độ chính xác bằng không. Sau đó, nếu một chuyển đổi với kiểu E có chỉ số mũ X:
* Nếu P > X >= -4, chuyển đổi sẽ được thực hiện theo kiểu f (hoặc F) và độ chính xác P - (X+1).
* Trong trường hợp khác, chuyển đổi sẽ được thực hiện theo kiểu e (hoặc E) và độ chính xác P - 1.
* Cuối cùng, trừ khi sử dụng cờ '#', các số không cuối cùng sẽ được loại bỏ khỏi phần thập phân của kết quả và ký tự dấu thập phân sẽ bị loại bỏ nếu không còn phần thập phân còn lại.

## PHẦN 21.2 HÀM PRINTF()

Hàm printf() là công cụ chính được sử dụng để in văn bản ra màn hình console trong C, được truy cập thông qua việc bao gồm thư viện <stdio.h>.

|  |
| --- |
| printf("Hello world!");  // Kết quả: Hello world! |

Chuỗi ký tự thông thường, không định dạng có thể được in bằng cách đặt chúng trực tiếp giữa hai dấu ngoặc đơn.

|  |
| --- |
| prntf("%d is the answer to life, the universe, and everything.", 42);  // 42 is the answer to life, the universe, and everything.  int x = 3;  char y = 'Z';  char\* z = "Example";  printf("Int: %d, Char: %c, String: %s", x, y, z);  // Int: 3, Char: Z, String: Example |

Định dạng các số nguyên, số thực, ký tự và nhiều loại dữ liệu khác có thể được in bằng cách sử dụng ký tự thoát **%**, sau đó là một ký tự hoặc chuỗi ký tự xác định định dạng, được gọi là bộ chuyển đổi định dạng.

Tất cả các đối số bổ sung cho hàm printf() được **phân tách bằng dấu phẩy** và các đối số này nên được sắp xếp theo thứ tự giống như các bộ chuyển đổi định dạng. Các đối số bổ sung sẽ bị bỏ qua, trong khi các đối số được nhập sai kiểu hoặc thiếu đối số sẽ gây ra lỗi hoặc hành vi không xác định. Mỗi đối số có thể là một giá trị chữ số hoặc một biến.

Sau khi thực thi thành công, hàm sẽ trả về số ký tự đã được in với kiểu int. Trong trường hợp thất bại, hàm trả về một giá trị số âm.

## PHẦN 21.3 CÁC CỜ ĐỊNH DẠNG KHI IN

Tiêu chuẩn C (C11 và C99) định nghĩa các cờ sau đây cho hàm printf():

|  |  |  |
| --- | --- | --- |
| **Flag(Cờ)** | **Conversions(Chuyển đổi)** | **Meaning(Ý nghĩa)** |
| - | Tất cả | Cờ "-" (dấu trừ) được sử dụng để căn trái kết quả của chuyển đổi trong trường. Nếu không có cờ này, chuyển đổi sẽ căn phải mặc định. |
| + | Số âm | Cờ "+" (dấu cộng) được sử dụng để luôn bắt đầu kết quả của chuyển đổi số có dấu với một dấu (+) hoặc (-). Chuyển đổi sẽ bắt đầu bằng dấu chỉ khi giá trị là âm nếu không có cờ này. |
| # | Tất cả | Cờ "#" (dấu hashtag) được sử dụng để chuyển đổi giá trị sang dạng thay thế. Cụ thể:  Đối với chuyển đổi "o", nếu cần thiết, cờ này sẽ tăng độ chính xác để buộc kết quả có chữ số đầu tiên là số 0 (nếu giá trị và độ chính xác đều là 0, chỉ có một số 0 được in).  Đối với chuyển đổi "x" hoặc "X", nếu giá trị không phải là 0, kết quả sẽ có tiền tố "0x" (hoặc "0X").  Đối với các chuyển đổi "a", "A", "e", "E", "f", "F", "g" và "G", kết quả luôn chứa một ký tự radix, ngay cả khi không có chữ số nào theo sau radix. Nếu không có cờ này, ký tự radix sẽ xuất hiện trong kết quả của các chuyển đổi này chỉ khi có một chữ số theo sau nó.  Đối với các chuyển đổi "g" và "G", các số 0 dư thừa không được loại bỏ khỏi kết quả như thường làm. Đối với các chuyển đổi khác, hành vi của cờ này là không xác định. |
| <space> | Số âm | khi sử dụng cờ <space> trong chuyển đổi có dấu, nếu kết quả không bắt đầu bằng dấu hoặc không có ký tự nào, một khoảng trắng sẽ được thêm vào đầu kết quả. Tuy nhiên, nếu cùng xuất hiện cùng với cờ +, thì cờ <space> sẽ bị bỏ qua. |
| 0 | Con số | Cờ "0" được sử dụng để thêm số 0 vào đầu các giá trị được chuyển đổi, để lấp đầy không gian trong trường hợp độ rộng trường nhỏ hơn độ dài của giá trị. Điều này chỉ áp dụng cho các chuyển đổi có kiểu "d", "i", "o", "u", "x", "X", "a", "A", "e", "E", "f", "F", "g" và "G".  Tuy nhiên, nếu xuất hiện cùng với cờ "-" (dấu trừ), cờ "0" sẽ bị bỏ qua. Điều này có nghĩa là nếu cả hai cờ "0" và "-" đều xuất hiện, cờ "0" sẽ không có tác dụng.  Đối với các chuyển đổi "d", "i", "o", "u", "x" và "X", nếu có độ chính xác được chỉ định, cờ "0" sẽ bị bỏ qua.  Nếu cùng với cờ "'" (dấu nháy đơn), các ký tự nhóm sẽ được chèn trước khi thêm số 0 vào đầu giá trị. Tuy nhiên, điều này chỉ áp dụng cho các chuyển đổi khác. Cho các chuyển đổi khác, hành vi của cờ "0" là không xác định. |

Các cờ này cũng được hỗ trợ bởi Microsoft với cùng ý nghĩa.

Bổ sung vào đó, theo tiêu chuẩn POSIX cho hàm printf(), có thêm một cờ là " **'** ", được áp dụng cho các chuyển đổi "i", "d", "u", "f", "F", "g" và "G".

|  |  |  |
| --- | --- | --- |
| **Flag(Cờ)** | **Conversions(Chuyển đổi)** | **Meaning(Ý nghĩa)** |
| ‘ | i, d, u, f, F, g, G | Khi sử dụng cờ " **'** ", phần nguyên của kết quả của một chuyển đổi thập phân sẽ được định dạng với các ký tự nhóm hàng nghìn. Điều này chỉ áp dụng cho các chuyển đổi liên quan đến số nguyên. |

## PHẦN 21.4 IN GIÁ TRỊ CỦA CON TRỎ TỚI ĐỐI TƯỢNG

Để in giá trị của một con trỏ tới một đối tượng (khác với con trỏ tới hàm), bạn sử dụng đặc tả chuyển đổi %p (conversion specifier %p). Đặc tả này được định nghĩa để in giá trị của các con trỏ kiểu void\*. Nếu bạn muốn in giá trị của một con trỏ kiểu khác (ví dụ: int\*, double\*, char\*, v.v.), bạn cần chuyển đổi ("casted\*") nó sang kiểu void\* trước khi sử dụng đặc tả %p.

|  |
| --- |
| #include <stdlib.h> /\* for EXIT\_SUCCESS \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {  int i;  int \* p = &i;  printf("The address of i is %p.\n", (void\*) p);  return EXIT\_SUCCESS;  } |

**Sử dụng <inttypes.h> và uintptr\_t**

Một cách khác để in con trỏ trong C99 trở lên sử dụng loại uintptr\_t và từ macro <inttypes.h>:

|  |
| --- |
| #include <inttypes.h> /\* for uintptr\_t and PRIXPTR \*/  #include <stdio.h> /\* for printf() \*/  int main(void)  {  int i;  int \* p = &i;  printf("The address of i is 0x%" PRIXPTR ".\n", (*uintptr\_t*)p);  return 0;  } |

Trong lý thuyết, có thể không có một kiểu số nguyên nào có thể chứa được bất kỳ con trỏ nào sau khi chuyển đổi thành một số nguyên (vì kích thước của con trỏ có thể lớn hơn kích thước của các kiểu số nguyên thông thường). Tuy nhiên, trong thực tế, kiểu uintptr\_t thường tồn tại và có khả năng chứa giá trị của con trỏ sau khi được chuyển đổi thành số nguyên.

Nếu kiểu uintptr\_t tồn tại, thì kiểu tương ứng intptr\_t cũng tồn tại. Kiểu uintptr\_t là một kiểu số nguyên không dấu có thể chứa giá trị của bất kỳ con trỏ nào, và kiểu intptr\_t là một kiểu số nguyên có dấu có thể chứa giá trị của bất kỳ con trỏ nào dưới dạng số nguyên có dấu.

**Lịch sử tiền tiêu chuẩn:**

Trước khi tiêu chuẩn C89 (K&R-C times), không tồn tại kiểu dữ liệu void\*, cũng không có thư viện tiêu chuẩn <stdlib.h>, không có các nguyên mẫu hàm, và do đó không có khai báo int main(void). Vì vậy, để in địa chỉ của một con trỏ, con trỏ được chuyển đổi sang kiểu long unsigned int và được in ra sử dụng đối tượng định dạng %lx (định dạng hexa dùng cho số nguyên dương dài).

**Tuy nhiên, cách làm này chỉ mang tính chất tham khảo và không hợp lệ trong các phiên bản C hiện đại như C89 trở lên, và có thể gây ra hành vi không xác định.**

|  |
| --- |
| #include <stdio.h> /\* optional in pre-standard C - for printf() \*/  int main()  {  int i;  int \*p = &i;  printf("The address of i is 0x%lx.\n", (long unsigned) p);  return 0;  } |

## PHẦN 21.5 IN GIÁ TRỊ CỦA HIỆU HAI CON TRỎ ĐẾN MỘT ĐỐI TƯỢNG

Khi lấy hiệu của hai con trỏ đến một đối tượng, kết quả là một số nguyên có dấu. Vì vậy, chúng ta nên in số này sử dụng ít nhất đối tượng định dạng %d.

Để đảm bảo rằng có một kiểu dữ liệu đủ rộng để chứa "hiệu của hai con trỏ", tiêu chuẩn C99 đã định nghĩa kiểu ptrdiff\_t. Để in giá trị của ptrdiff\_t, chúng ta sử dụng đối tượng định dạng %td.

|  |
| --- |
| #include <stdlib.h> /\* for EXIT\_SUCCESS \*/  #include <stdio.h> /\* for printf() \*/  #include <stddef.h> /\* for ptrdiff\_t \*/  int main(void)  {  int a[2];  int \*p1 = &a[0], \*p2 = &a[1];  *ptrdiff\_t* pd = p2 - p1;  printf("p1 = %p\n", (void\*) p1);  printf("p2 = %p\n", (void\*) p2);  printf("p2 - p1 = %td\n", pd);  return EXIT\_SUCCESS;  } |

Kết quả có thể nhìn thấy như sau

|  |
| --- |
| p1 = 0061FF0C  p2 = 0061FF10  p2 - p1 = 1 |

Lưu ý rằng giá trị kết quả của hiệu hai con trỏ được **tỉ lệ** bởi kích thước của kiểu dữ liệu mà con trỏ trừ trỏ tới, trong ví dụ này là kiểu int. Kích thước của một kiểu int trong ví dụ này là 4 byte.

Chú ý: Nếu hai con trỏ được trừ không trỏ tới cùng một đối tượng, thì hành vi của chương trình sẽ không xác định (undefined behavior).

## PHẦN 21.6 LENGTH MODIFIERS

Phần này trình bày về các "length modifiers" trong hàm printf() theo tiêu chuẩn C99 và C11. Chúng được sử dụng để chỉ định kiểu dữ liệu của đối số truyền vào hàm printf(). Dưới đây là danh sách các length modifiers và ý nghĩa của chúng:

Các length modifiers tiêu chuẩn (theo tiêu chuẩn C99 và C11):

|  |  |  |
| --- | --- | --- |
| Modifier | Modifies | Applies to |
| hh | d, i, o, u, x, or X | char, signed char or unsigned char |
| h | d, i, o, u, x, or X | short int or unsigned short int |
| l | d, i, o, u, x, or X | long int or unsigned long int |
| l | a, A, e, E, f, F, g, or G | double (for compatibility with scanf(); undefined in C90) |
| ll | d, i, o, u, x, or X | long long int or unsigned long long int |
| j | d, i, o, u, x, or X | intmax\_t or uintmax\_t |
| z | d, i, o, u, x, or X | size\_t or the corresponding signed type (ssize\_t in POSIX) |
| t | d, i, o, u, x, or X | ptrdiff\_t or the corresponding unsigned integer type |
| L | a, A, e, E, f, F, g, or G | long double |

Đối với những length modifier mà không được định nghĩa trong tiêu chuẩn C99 và C11, việc sử dụng chúng với các conversion specifier khác sẽ dẫn đến hành vi không xác định trong ngôn ngữ lập trình C.

Microsoft đặc tả một số length modifiers khác nhau mà không thuộc tiêu chuẩn C, và họ rõ ràng không hỗ trợ hh, j, z, hoặc t.

|  |  |  |
| --- | --- | --- |
| Modifier | Modifies | Applies to |
| I32 | d, i, o, x, or X | \_\_int32 |
| I32 | o, u, x, or X | unsigned \_\_int32 |
| I64 | d, i, o, x, or X | \_\_int64 |
| I64 | o, u, x, or X | unsigned \_\_int64 |
| I | d, i, o, x, or X | ptrdiff\_t (that is, \_\_int32 on 32-bit platforms, \_\_int64 on 64-bit platforms) |
| I | o, u, x, or X | size\_t (that is, unsigned \_\_int32 on 32-bit platforms, unsigned \_\_int64 on 64-bit platforms) |
| l or L | a, A, e, E, f, g, or G | long double (In Visual C++, although long double is a distinct type, it has the same internal representation as double.) |
| l or w | c or C | Wide character with printf and wprintf functions. (An lc, lC, wc or wC type specifier is synonymous with C in printf functions and with c in wprintf functions.) |

Lưu ý rằng các bộ chỉ định chuyển đổi C, S và Z cũng như các bộ sửa đổi độ dài I, I32, I64 và w là các phần mở rộng của Microsoft. Việc coi l như một công cụ sửa đổi cho long double thay vì double khác với tiêu chuẩn, mặc dù bạn sẽ khó phát hiện ra sự khác biệt trừ khi long double có cách biểu diễn khác với double.

# CON TRỎ

Con trỏ là một loại biến có thể lưu trữ địa chỉ của một **đối tượng** hoặc một **hàm**.

## Giới thiệu

Một con trỏ được khai báo tương tự như bất kỳ biến nào khác, chỉ có một dấu **\*** được đặt giữa kiểu dữ liệu và tên của biến để chỉ định rằng nó là một **con trỏ**.

|  |
| --- |
| /\* Trong một hàm, con trỏ chưa được khởi tạo và không trỏ đến bất kỳ đối tượng hợp lệ nào \*/  int \*pointer; |

Để khai báo hai biến con trỏ cùng kiểu trong cùng một khai báo, sử dụng ký hiệu \* trước mỗi định danh. Ví dụ:

|  |
| --- |
| int \*iptr1, \*iptr2;  int \*iptr3, iptr4;  /\* iptr3 là một biến con trỏ, trong khi iptr4 là một biến kiểu int \*/ |

Toán tử **địa chỉ** hoặc **tham chiếu** được ký hiệu bằng dấu **&** cho phép lấy địa chỉ của một biến cụ thể, sau đó địa chỉ này có thể được đặt trong một con trỏ có kiểu thích hợp.

|  |
| --- |
| int value = 1;  pointer = &value; |

Toán tử indirection hoặc dereference được ký hiệu bằng dấu **\*** cho phép **Truy cập nội dung** của một đối tượng được trỏ đến bởi một con trỏ.

|  |
| --- |
| printf("Value of pointed to integer: %d\n", \*pointer);  /\* Value of pointed to integer: 1 \*/ |

Nếu con trỏ trỏ đến kiểu structure hoặc union, bạn có thể **dereference** nó và truy cập trực tiếp các thành viên của nó bằng toán tử **->**:

|  |
| --- |
| SomeStruct \*s = &someObject;  s->someMember = 5; /\* Equivalent to (\*s).someMember = 5 \*/ |

Trong C, một con trỏ là một loại giá trị riêng biệt có thể được gán lại và được xem xét như một biến riêng biệt. Ví dụ, ví dụ sau đây in giá trị của con trỏ (biến) chính nó.

|  |
| --- |
| printf("Gia tri cua con tro: %p\n", (void \*)pointer);  /\* Giá trị của con trỏ: 0x7ffcd41b06e4 \*/  /\* Địa chỉ này sẽ khác nhau mỗi khi chương trình được thực thi \*/ |

Bởi vì con trỏ là một biến có thể thay đổi, nó có thể không trỏ đến một đối tượng hợp lệ, thông qua việc được gán giá trị null.

Hoặc chỉ đơn giản chứa một mẫu bit tùy ý không phải là địa chỉ hợp lệ. Trường hợp sau là một tình huống rất tồi tệ, vì nó không thể kiểm tra trước khi con trỏ được **dereference**, chỉ có một kiểm tra cho trường hợp con trỏ là null:

|  |
| --- |
| if (!pointer) exit(EXIT\_FAILURE); |

Một con trỏ chỉ có thể **dereferenced** nếu nó trỏ đến một đối tượng **hợp lệ**, nếu không thì hành vi của nó là **không xác định**. Nhiều phiên bản hiện đại có thể giúp bạn bằng cách tạo ra một lỗi nào đó như lỗi **segmentation fault** và kết thúc thực thi, nhưng những phiên bản khác có thể chỉ để lại chương trình của bạn trong trạng thái không hợp lệ.

Giá trị trả về từ toán tử **dereference** là một bí danh có thể thay đổi của biến gốc, vì vậy nó có thể được thay đổi, làm **thay đổi biến gốc**.

|  |
| --- |
| \*pointer += 1;  printf("Gia tri cua bien ma con tro tro toi sau khi thay doi la: %d\n", \*pointer);  /\*Giá trị của biến mà con trỏ trỏ tới sau khi thay đổi là 2\*/ |

Con trỏ cũng có thể được gán lại. Điều này có nghĩa là một con trỏ trỏ đến một đối tượng sau này có thể được sử dụng để **trỏ đến một đối tượng khác** cùng kiểu.

|  |
| --- |
| int value2 = 10;  pointer = &value2;  printf("Gia tri tu con tro: %d \n", \*pointer);  /\*Giá trị từ con trỏ: 10 \*/ |

Giống như bất kỳ biến nào khác, con trỏ có một kiểu cụ thể. Bạn không thể gán địa chỉ của một **short int** cho một con trỏ kiểu **long int**, ví dụ. Hành vi như vậy được gọi là type punning và bị cấm trong C, mặc dù có một số ngoại lệ.

Mặc dù con trỏ phải có một kiểu cụ thể, bộ nhớ được cấp phát cho mỗi loại con trỏ bằng với bộ nhớ được sử dụng bởi **môi trường để lưu trữ các địa chỉ**, thay vì kích thước của kiểu dữ liệu được trỏ tới.

|  |
| --- |
| #include <stdio.h>  int main(void) {  printf("Size of int pointer: %zu\n", sizeof (int\*)); /\* size 4 bytes \*/  printf("Size of int variable: %zu\n", sizeof (int)); /\* size 4 bytes \*/  printf("Size of char pointer: %zu\n", sizeof (char\*)); /\* size 4 bytes \*/  printf("Size of char variable: %zu\n", sizeof (char)); /\* size 1 bytes \*/  printf("Size of short pointer: %zu\n", sizeof (short\*)); /\* size 4 bytes \*/  printf("Size of short variable: %zu\n", sizeof (short)); /\* size 2 bytes \*/  return 0;  } |

Như vậy, chúng ta thấy rằng khi chạy trên nền tảng hệ điều hành 32 bits con trỏ sẽ có kích thước 4 bytes, khi chạy trên nền tảng hệ điều hành 64 bits con trỏ sẽ có kích thước 8 bytes.

Kiểu dữ liệu của con trỏ thay đổi **không hề tác động** đến kích thước bộ nhớ của con trỏ. Bởi vì giá trị thực sự của con trỏ là kiểu số nguyên không dấu (unsigned int), trong nền tảng hệ điều hành 32 bits, giá trị mà con trỏ lưu trữ sẽ là ***unsigned \_\_int32***, và trong nền tảng hệ điều hành 64 bits, giá trị của con trỏ lưu trữ có kiểu ***unsigned \_\_int64***.

Phạm vi giá trị của ***unsigned \_\_int32*** là từ 0 đến 4294967295, phạm vi giá trị này đủ để đánh dấu địa chỉ tất cả ô nhớ trong **Virtual memory** của nền tảng hệ điều hành **32 bits**. Tương tự, phạm vi giá trị của ***unsigned \_\_int64*** là tử 0 đến 18446744073709551615, đủ để đánh dấu địa chỉ của tất cả ô nhớ trong **Virtual memory** của nền tảng hệ điều hành **64 bits**.

(NB: Nếu bạn đang sử dụng Microsoft Visual Studio, không hỗ trợ các tiêu chuẩn C99 hoặc C11, bạn phải sử dụng %Iu1 thay vì %zu trong ví dụ trên.)

Lưu ý rằng các kết quả ở trên có thể khác nhau giữa các môi trường nhưng tất cả các môi trường sẽ hiển thị kích thước bằng nhau cho các loại con trỏ khác nhau.

Trích xuất dựa trên thông tin từ [Cardiff University C Pointers Introduction](https://users.cs.cf.ac.uk/Dave.Marshall/C/node10.html)

**Con trỏ và mảng**

Con trỏ và mảng liên kết chặt chẽ trong C. Mảng trong C luôn được lưu giữ trong các vị trí liền kề trong bộ nhớ. Toán tử tính toán con trỏ luôn được tỉ lệ theo kích thước của mục được trỏ đến. Vì vậy, nếu chúng ta có một mảng ba số double và một con trỏ trỏ tới cơ sở, **\*ptr** tham chiếu đến số double đầu tiên, **\*(ptr + 1)** tham chiếu đến số double thứ hai, **\*(ptr + 2)** tham chiếu đến số double thứ ba. Một cách ghi chú tiện ích hơn là sử dụng ký hiệu mảng [].

|  |
| --- |
| double point[3] = {0.0, 1.0, 2.0};  double \*ptr = point;  /\* prints x 0.0, y 1.0 z 2.0 \*/  printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]); |

Vì vậy, về cơ bản **ptr** và **tên mảng** có thể hoán đổi cho nhau. Quy tắc này cũng có nghĩa là một mảng phân rã thành một con trỏ khi được chuyển đến một chương trình con.

|  |
| --- |
| double point[3] = {0.0, 1.0, 2.0};  printf("length of point is %s\n", length(*point*));  /\* get the distance of a 3D point from the origin \*/  double length(double \**pt*)  {  return sqrt(*pt*[0] \* *pt*[0] + *pt*[1] \* *pt*[1] + *pt*[2] \* *pt*[2])  } |

Một con trỏ có thể trỏ đến bất kỳ phần tử nào trong mảng, hoặc đến phần tử ngoài cùng của mảng. Tuy nhiên, là một lỗi khi đặt một con trỏ thành bất kỳ giá trị nào khác, bao gồm cả phần tử trước mảng. (Lý do là trên kiến trúc phân đoạn địa chỉ trước phần tử đầu tiên có thể vượt qua ranh giới phân đoạn, trình biên dịch đảm bảo rằng điều đó không xảy ra cho phần tử cuối cùng cộng thêm một).

Chú thích cuối trang 1: Có thể tìm thấy thông tin định dạng của Microsoft qua printf() và cú pháp đặc tả định dạng.

## Các lỗi thường gặp

Việc sử dụng con trỏ không đúng cách thường là nguồn gốc của lỗi có thể bao gồm lỗi bảo mật hoặc sự cố chương trình, thường là do lỗi phân đoạn.

Lỗi phân đoạn là một loại lỗi cụ thể do truy cập bộ nhớ mà không thuộc về bạn. Đây là cơ chế trợ giúp giúp bạn không làm hỏng bộ nhớ và đưa ra các lỗi bộ nhớ khó gỡ lỗi. Bất cứ khi nào bạn gặp lỗi segfault, bạn biết rằng mình đang làm gì đó với bộ nhớ - truy cập biến đã được giải phóng, ghi vào phần chỉ đọc của bộ nhớ, v.v. Lỗi phân đoạn về cơ bản là giống nhau trong hầu hết các ngôn ngữ cho phép bạn gây rối việc quản lý bộ nhớ

### Không kiểm tra lỗi trong việc cấp phát bộ nhớ

Việc cấp phát bộ nhớ không đảm bảo sẽ thành công và có thể trả về một con trỏ **NULL**. Sử dụng giá trị trả về mà không kiểm tra xem việc cấp phát thành công hay không sẽ gây ra hành vi không xác định. Điều này thường dẫn đến sự cố của chương trình, nhưng không có đảm bảo rằng sự cố sẽ xảy ra, vì vậy dựa vào điều đó cũng có thể gây ra vấn đề.

**Ví dụ không an toàn**

|  |
| --- |
| struct *SomeStruct* \*s = malloc(sizeof \**s*);  s->someValue = 0; /\* KHÔNG AN TOÀN, vì s có thể là con trỏ null \*/ |

**Cách an toàn**

|  |
| --- |
| struct *SomeStruct* \*s = malloc(sizeof \**s*);  if (s){  s->someValue = 0; /\* Điều này là an toàn, chúng ta đã kiểm tra rằng s là hợp lệ \*/  } |

### Sử dụng số nguyên cố định thay vì sizeof khi yêu cầu cấp phát bộ nhớ

Đối với một cấu hình trình biên dịch/máy tính cụ thể, các kiểu dữ liệu có kích thước đã biết; tuy nhiên, không có tiêu chuẩn nào xác định kích thước của một kiểu dữ liệu cụ thể (ngoại trừ char) sẽ giống nhau trên tất cả các cấu hình trình biên dịch/máy tính. Nếu mã sử dụng số 4 thay vì sizeof(int) để cấp phát bộ nhớ, nó có thể hoạt động trên máy tính ban đầu, nhưng mã không nhất thiết là có thể chạy trên các máy tính hoặc trình biên dịch khác nhau. Các kích thước cố định cho các kiểu dữ liệu nên được thay thế bằng

|  |
| --- |
| sizeof(that\_type) hoặc sizeof(\*var\_ptr\_to\_that\_type) |

Cấp phát không di động

|  |
| --- |
| int \*intPtr = malloc(4\*1000); /\* cấp phát bộ nhớ cho 1000 int \*/  long \*longPtr = malloc(8\*1000); /\* cấp phát bộ nhớ cho 1000 long \*/ |

Cấp phát động

|  |
| --- |
| int \*intPtr = malloc(sizeof(int)\*1000); /\* cấp phát bộ nhớ cho 1000 int \*/  long \*longPtr = malloc(sizeof(long)\*1000); /\* cấp phát bộ nhớ cho 1000 long \*/ |

Hoặc tốt hơn cả:

|  |
| --- |
| int \*intPtr = malloc(sizeof(\*intPtr)\*1000); /\* cấp phát bộ nhớ cho 1000 int \*/  long \*longPtr = malloc(sizeof(\*longPtr)\*1000); /\* cấp phát bộ nhớ cho 1000 long \*/ |

### Rò rỉ bộ nhớ (Memory leaks)

Không giải phóng bộ nhớ bằng cách sử dụng **free** dẫn đến tích lũy bộ nhớ không thể tái sử dụng, mà không còn được sử dụng bởi chương trình; điều này được gọi là rò rỉ bộ nhớ (**memory leak**). Rò rỉ bộ nhớ lãng phí tài nguyên bộ nhớ và có thể dẫn đến lỗi cấp phát.

### Lỗi Logic

Tất cả các cấp phát đều phải tuân theo mẫu sau:

1. Cấp phát bằng **malloc** (hoặc **calloc**).
2. Sử dụng để lưu trữ dữ liệu.
3. Giải phóng bằng **free**.

Không tuân theo mẫu này, chẳng hạn như sử dụng bộ nhớ sau khi gọi free (dangling pointer) con trỏ treo hoặc trước khi gọi malloc (wild pointer) con trỏ hoang, gọi free hai lần ("double free"), vv., thường dẫn đến lỗi phân đoạn và dẫn đến sự cố của chương trình.

Những lỗi này có thể tạm thời và khó debug - ví dụ: bộ nhớ đã được giải phóng thường không được thu hồi ngay lập tức bởi hệ điều hành, do đó dangling pointer (con trỏ treo) có thể tồn tại trong một thời gian và có thể hoạt động bình thường.

Trên các hệ thống mà nó hoạt động, Valgrind là một công cụ quý giá để xác định bộ nhớ bị rò rỉ và nơi nó được cấp phát ban đầu.

### Tạo con trỏ trỏ tới biến được tạo trong vùng nhớ (stack) ngăn xếp

Việc tạo một con trỏ trỏ tới biến sẽ bị xóa đi(biến cục bộ) . Ví dụ:

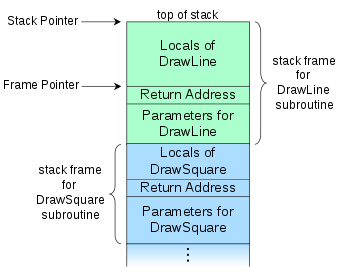
|  |
| --- |
| int\* myFunction()  {  int x = 10;  return &x;  } |

Ở đoạn mã trên, biến x có hệ thông lưu trữ tự động (thường được gọi là cấp phát trên vùng nhớ stack (ngăn xếp). Vì nó được cấp phát trên vùng ngăn xếp, thời gian tồn tại của biến x chỉ kéo dài trong khi hàm **myFunction** đang thực thi; sau khi hàm **myFunction** kết thúc, biến x bị **xóa**. Hàm **myFunction** lấy địa chỉ của biến x (sử dụng &x) và trả về nó cho hàm gọi, khiến cho hàm gọi có một con trỏ trỏ đến **một biến không tồn tại**. Cố gắng truy cập vào biến này sẽ gây ra hành vi không xác định.

Tuy nhiên, hầu hết các trình biên dịch thực tế không xóa **stack frame** sau khi hàm thoát, do đó, khi truy cập con trỏ được trả về, thường bạn vẫn nhận được dữ liệu mong đợi. Tuy nhiên, khi gọi một hàm khác, bộ nhớ mà con trỏ đang trỏ đến có thể bị ghi đè và dữ liệu trong bộ nhớ có thể bị hỏng.

**stack frame**

Ngăn xếp (stack) là một phân đoạn (segment) của bộ nhớ (memory) tăng trưởng và thu hẹp một cách động có chứa các stack frame. Một stack frame được cấp phát cho mỗi hàm (function) hiện đang được gọi. Stack frame lưu trữ các biến cục bộ (local variable) của hàm, đối số (argument) và giá trị trả về (return value). Một stack frame được thêm vào ngăn xếp stack mỗi khi một hàm mới được gọi và loại bỏ khi hàm đó được thoát / trả về (return). Một ngăn xếp bao gồm một tập hợp các stack frame được quản lý theo cơ chế vào trước ra sau



Để giải quyết vấn đề này, bạn có thể cấp phát bộ nhớ động cho biến cần trả về bằng cách sử dụng malloc và trả về con trỏ đến bộ nhớ mới được tạo, hoặc bạn có thể yêu cầu đối số là một con trỏ hợp lệ được truyền vào hàm thay vì trả về một con trỏ.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  int \*solution1(void)  {  int \*x = malloc(sizeof \*x);  if (x == NULL)  {  /\* Something went wrong \*/  return NULL;  }  \*x = 10;  return x;  }  void solution2(int \**x*)  {  /\* Lưu ý: gọi hàm này với một con trỏ không hợp lệ hoặc null sẽ gây ra hành vi không xác định. \*/  \**x* = 10;  }  int main(void)  {  {  /\* Sử dụng solution1() \*/  int \*foo = solution1();  if (foo == NULL)  {  /\* Đã xảy ra lỗi \*/  return 1;  }  printf("Giá trị được đặt bởi solution1() là %i\n", \*foo);  /\* Sẽ in ra: "Giá trị được đặt bởi solution1() là 10" \*/  free(foo); /\* Dọn dẹp \*/  }  {  /\* Sử dụng solution2() \*/  int bar;  solution2(&bar);  printf("Giá trị được đặt bởi solution2() là %i\n", bar);  /\* Sẽ in ra: "Giá trị được đặt bởi solution2() là 10" \*/  }  return 0;  } |

### Tăng giảm và dereferencing

Nếu bạn viết \*p++ để tăng giá trị mà p trỏ tới, bạn sai.

Tăng / giảm được thực thi trước dereferencing. Do đó, biểu thức \*p++ sẽ tự tăng con trỏ p và trả về giá trị được trỏ bởi p trước khi tăng mà không thay đổi nó.

Ví dụ: nếu p trỏ đến số 10, thì \*p++ sẽ trả về số 10 và sau đó tăng p lên một đơn vị. Do đó, p sẽ trỏ đến số 11 **sau khi biểu thức được thực thi**.

Bạn có thể nghĩ về biểu thức \*p++ như sau:

1. Truy cập giá trị được trỏ bởi p.
2. Tăng p lên một đơn vị.
3. Trả về giá trị được truy cập ở bước 1.

Bạn nên viết (\*p)++ để tăng giá trị mà p trỏ tới.

Quy tắc này cũng áp dụng cho giảm sau: \*p-- sẽ giảm giá trị của con trỏ p chính nó, không phải giá trị mà p trỏ tới.

## Dereferencing a pointer

Dereferencing trong lập trình được sử dụng để truy cập hoặc thao tác dữ liệu có trong vị trí bộ nhớ (memory location) được trỏ tới bởi một con trỏ (pointer). \* (dấu hoa thị) được sử dụng với con trỏ khi dereferencing con trỏ, nó đề cập đến biến được trỏ, vì vậy đây được gọi là dereferencing đến con trỏ.

|  |
| --- |
| int a = 1;  int \*a\_pointer = &a; |

Để dereference a\_pointer và thay đổi giá trị của a, chúng ta sử dụng thao tác sau

|  |
| --- |
| \*a\_pointer = 2; |

Điều này có thể được xác minh bằng cách sử dụng các câu lệnh in sau đây.

|  |
| --- |
| printf("%d\n", a); /\* Prints 2 \*/  printf("%d\n", \**a\_pointer*); /\* Also prints 2 \*/ |

Tuy nhiên, người ta sẽ nhầm lẫn "to dereference" một con trỏ NULL hoặc con trỏ không hợp lệ. Cái này:

|  |
| --- |
| int \*p1, \*p2;  p1 = (int \*) 0xbad;  p2 = NULL;  \*p1 = 42;  \*p2 = \*p1 + 1; |

thường là hành vi không xác định . . p1 có thể không bị "dereference" vì nó trỏ đến địa chỉ 0xbad có thể không phải là địa chỉ hợp lệ. Ai biết có gì ở đó? Nó có thể là bộ nhớ của hệ điều hành hoặc bộ nhớ của chương trình khác. Mã thời gian duy nhất như thế này được sử dụng là trong quá trình phát triển nhúng, nơi lưu trữ thông tin cụ thể tại các địa chỉ được mã hóa cứng. p2 không thể hủy đăng ký vì nó là NULL, không hợp lệ.

## Dereferencing a pointer to a struct

Giả sử chúng ta có cấu trúc sau đây

|  |
| --- |
| struct *MY\_STRUCT*  {  int my\_int;  float my\_float;  }; |

Chúng ta có thể định nghĩa MY\_STRUCT mà không cần sử dụng từ khóa "struct" mỗi lần sử dụng nó. Điều này tuy nhiên là tùy chọn.

|  |
| --- |
| typedef struct *MY\_STRUCT* *MY\_STRUCT*; |

Nếu chúng ta có một con trỏ tới một phiên bản của cấu trúc này:

|  |
| --- |
| *MY\_STRUCT* \*instance; |

Nếu câu lệnh này xuất hiện ở file scope, thì instance sẽ được khởi tạo với một con trỏ null khi chương trình bắt đầu. Nếu câu lệnh này xuất hiện trong một hàm, giá trị của nó là không xác định. Biến này phải được khởi tạo để trỏ tới một biến MY\_STRUCT hợp lệ hoặc đến không gian được cấp phát động trước khi có thể “dereference” nó. Ví dụ:

|  |
| --- |
| *MY\_STRUCT* info = { 1, 3.141593F };  *MY\_STRUCT* \*instance = &info; |

Khi con trỏ là hợp lệ, chúng ta có thể “dereference” nó để truy cập các thành viên của nó bằng hai cách khác nhau:

|  |
| --- |
| int a = (\*instance).my\_int;  float b = instance->my\_float; |

Mặc dù cả hai cách này đều hoạt động, tuy nhiên việc sử dụng toán tử mũi tên -> là tốt hơn so với việc sử dụng sự kết hợp của dấu ngoặc đơn, toán tử dereference \* và toán tử dấu chấm . bởi vì nó dễ đọc và hiểu hơn, đặc biệt là với các cấu trúc lồng nhau.

Một điểm khác biệt quan trọng khác được thể hiện dưới đây:

|  |
| --- |
| *MY\_STRUCT* copy = \*instance;  copy.my\_int = 2; |

Trong trường hợp này, biến copy chứa một bản sao nội dung của instance. Thay đổi my\_int của copy sẽ không thay đổi my\_int của instance.

|  |
| --- |
| *MY\_STRUCT* \*ref = instance;  ref->my\_int = 2; |

Trong trường hợp này, ref là một tham chiếu đến instance. Thay đổi my\_int sử dụng tham chiếu sẽ thay đổi my\_int trong instance.

Thường thì chúng ta sử dụng con trỏ tới các cấu trúc như các tham số trong các hàm, thay vì truyền trực tiếp các cấu trúc. Sử dụng các cấu trúc như các tham số của hàm có thể làm tràn ngăn xếp (stack overflow) nếu cấu trúc lớn. Sử dụng con trỏ tới cấu trúc chỉ sử dụng đủ không gian ngăn xếp cho con trỏ, nhưng có thể gây ra tác động phụ nếu hàm thay đổi cấu trúc được truyền vào hàm.

## Con trỏ hằng

### Single Pointers

* Pointer to an int (Con trỏ trỏ tới một số nguyên)

Con trỏ này có thể trỏ tới các số nguyên khác nhau và giá trị của số nguyên có thể được thay đổi thông qua con trỏ. Đoạn mã ví dụ sau gán con trỏ 'p' trỏ tới số nguyên 'b', sau đó thay đổi giá trị của 'b' thành 100.

|  |
| --- |
| int b;  int\* p;  p = &b; /\* Được phép \*/  \*p = 100; /\* Được phép \*/  /\*Lúc này giá trị b = 100\*/ |

* Pointer to a const int (Hằng con trỏ int)

Con trỏ này có thể trỏ tới các số nguyên khác nhau, nhưng giá trị của số nguyên **không thể thay đổi** thông qua con trỏ.

Hằng con trỏ là con trỏ trỏ đến một địa chỉ cố định, giá trị của con trỏ không được phép thay đổi, nhưng **có thể thay đổi giá trị của vùng nhớ** mà con trỏ trỏ đến.

|  |
| --- |
| int b;  const int\* p;  p = &b; /\* Được phép \*/  \*p = 100; /\* Lỗi biên dịch \*/ |

* const pointer to int (Con trỏ hằng int)

Con trỏ này chỉ có thể trỏ tới một số nguyên và giá trị của số nguyên có thể được thay đổi thông qua con trỏ.

Con trỏ hằng là con trỏ trỏ đến một **địa chỉ mà không thể thay đổi giá trị** lưu ở địa chỉ mà nó trỏ đến, nhưng việc thay đổi **giá trị** của con trỏ **vẫn hợp lệ**.

* const pointer to const int (Hằng con trỏ hằng) (Sử dụng kết hợp con trỏ hằng và hằng con trỏ)

Con trỏ này chỉ có thể trỏ tới một số nguyên và giá trị của số nguyên không thể thay đổi thông qua con trỏ.

Sử dụng kết hợp cả 2 tính năng trên sẽ được 1 kết quả là con trỏ không được phép thay đổi giá trị của nó lẫn giá trị vùng nhớ mà nó đang trỏ đến.

|  |
| --- |
| int a, b;  const int\* const p = &b; /\* Được phép khi khởi tạo, không thể gán lại sau đó \*/  p = &a; /\* Lỗi biên dịch \*/  \*p = 100; /\* Lỗi biên dịch \*/ |

### Pointer to Pointer

* Pointer to a pointer to an int (Con trỏ trỏ tới con trỏ trỏ tới một số nguyên)

Đoạn mã này gán địa chỉ của 'p1' cho con trỏ hai lần 'p' (mà sau đó trỏ tới con trỏ 'p1' (nó lại trỏ tới số nguyên)).

Sau đó thay đổi 'p1' để trỏ tới số nguyên 'a'. Sau đó thay đổi giá trị của 'a' thành 100.

|  |
| --- |
| void f1(void)  {  int a, b;  int \*p1;  int \*\*p;  p1 = &b; /\* Được phép \*/  p = &p1; /\* Được phép \*/  \*p = &a; /\* Được phép \*/  \*\*p = 100; /\* Được phép \*/  } |

* Pointer to pointer to a const int

|  |
| --- |
| void f2(void)  {  int b;  const int \*p1;  const int \*\*p;  p = &p1; /\* Được phép \*/  \*p = &b; /\* Được phép \*/  \*\*p = 100; /\* Lỗi: Gán giá trị cho vị trí chỉ đọc ‘\*\*p’ \*/  } |

* Pointer to const pointer to an int

|  |
| --- |
| void f3(void)  {  int b;  int \*p1;  int \* const \*p;  p = &p1; /\* Được phép \*/  \*p = &b; /\* Lỗi: Gán giá trị cho vị trí chỉ đọc ‘\*p’ \*/  \*\*p = 100; /\* Được phép \*/  } |

* const pointer to pointer to int

|  |
| --- |
| void f4(void)  {  int b;  int \*p1;  int \*\* const p = &p1; /\* Được phép khi khởi tạo, không thể gán lại sau đó \*/  p = &p1; /\* Lỗi: Gán biến chỉ đọc ‘p’ \*/  \*p = &b; /\* Được phép \*/  \*\*p = 100; /\* Được phép \*/  } |

* Pointer to const pointer to const int

|  |
| --- |
| void f5(void)  {  int b;  const int \*p1;  const int \* const \*p;  p = &p1; /\* Được phép \*/  \*p = &b; /\* Lỗi: Gán giá trị cho vị trí chỉ đọc ‘\*p’ \*/  \*\*p = 100; /\* Lỗi: Gán giá trị cho vị trí chỉ đọc ‘\*\*p’ \*/  } |

* const pointer to pointer to const int

|  |
| --- |
| void f6(void)  {  int b;  const int \*p1;  const int \*\* const p = &p1; /\* Được phép khi khởi tạo, không thể gán lại sau đó \*/  p = &p1; /\* Lỗi: Gán biến chỉ đọc ‘p’ \*/  \*p = &b; /\* Được phép \*/  \*\*p = 100; /\* Lỗi: Gán giá trị cho vị trí chỉ đọc ‘\*\*p’ \*/  } |

* const pointer to const pointer to int

|  |
| --- |
| void f7(void)  {  int b;  int \*p1;  int \* const \* const p = &p1; /\* Được phép khi khởi tạo, không thể gán lại sau đó \*/  p = &p1; /\* Lỗi: Gán biến chỉ đọc ‘p’ \*/  \*p = &b; /\* Lỗi: Gán giá trị cho vị trí chỉ đọc ‘\*p’ \*/  \*\*p = 100; /\* Được phép \*/  } |

## Con trỏ hàm

Con trỏ cũng có thể được sử dụng để trỏ tới các hàm.

Vì function nằm ở code segment, là vùng nhớ read-only khi thực thi chương trình nên function pointer là một pointer to const

Chúng ta sẽ lấy một hàm cơ bản làm ví dụ:

|  |
| --- |
| int my\_function(int *a*, int *b*)  {  return 2 \* *a* + 3 \* *b*;  } |

Tiếp theo, chúng ta xác định một con trỏ có kiểu tương tự với hàm này:

|  |
| --- |
| int (\*my\_pointer)(int, int); |

Để tạo một con trỏ như vậy, bạn chỉ cần sử dụng mẫu sau:

|  |
| --- |
| kiểu\_trả\_về\_của\_hàm (\*con\_trỏ\_hàm)(kiểu\_tham\_số1, kiểu\_tham\_số2, ...) |

Sau đó, chúng ta phải gán con trỏ này tới hàm:

|  |
| --- |
| my\_pointer = &my\_function; |

Con trỏ này có thể được sử dụng để gọi hàm:

|  |
| --- |
| /\* Gọi hàm được trỏ tới \*/  int result = (\*my\_pointer)(4, 2);  ...  /\* Sử dụng con trỏ hàm như một tham số cho một hàm khác \*/  void another\_function(int (\*another\_pointer)(int, int))  {  int a = 4;  int b = 2;  int result = (\*another\_pointer)(a, b);  printf("%d\n", result);  } |

Mặc dù cú pháp này dường như rõ ràng và nhất quán với các kiểu cơ bản, nhưng việc gán và dereferencing con trỏ hàm không đòi hỏi sử dụng các toán tử & và \*. Vì vậy, đoạn mã sau đây cũng hoàn toàn hợp lệ:

|  |
| --- |
| /\* Gán giá trị mà không sử dụng toán tử & \*/  my\_pointer = my\_function;  function name trả về địa chỉ của function Tương tự như tên của array trả về địa chỉ của mảng  /\* Dereferencing mà không sử dụng toán tử \* \*/  int result = my\_pointer(4, 2); |

Để tăng tính đọc hiểu của con trỏ hàm, bạn có thể sử dụng typedef.

|  |
| --- |
| typedef void (\**Callback*)(int a);  void some\_function(*Callback* *callback*)  {  int a = 4;  *callback*(a);  } |

Một mẹo đọc hiểu khác là trong tiêu chuẩn C, bạn có thể đơn giản hóa con trỏ hàm trong đối số như trên (nhưng không phải trong khai báo biến) thành một cái gọi là một nguyên mẫu hàm; do đó bạn có thể sử dụng cú pháp sau đây cho việc định nghĩa và khai báo hàm:

|  |
| --- |
| void some\_function(void *callback*(int))  {  int a = 4;  *callback*(a);  } |

Xem thêm

Con trỏ hàm (Function Pointers)

## Polymorphic behaviour with void pointers

Hàm thư viện tiêu chuẩn qsort() là một ví dụ tốt về cách sử dụng con trỏ void để thực hiện một hàm duy nhất trên nhiều loại dữ liệu khác nhau.

Hàm qsort() có dạng như sau:

|  |
| --- |
| void qsort(  void \**base*, /\* Mảng cần sắp xếp \*/  *size\_t* *num*, /\* Số phần tử trong mảng \*/  *size\_t* *size*, /\* Kích thước của mỗi phần tử (tính bằng byte) \*/  int (\**compar*)(const void \*, const void \*)); /\* Hàm so sánh hai phần tử \*/ |

Mảng cần được sắp xếp được truyền dưới dạng con trỏ void, vì vậy một mảng của bất kỳ kiểu dữ liệu nào cũng có thể được sử dụng. Hai đối số tiếp theo cho biết qsort() cần mong đợi bao nhiêu phần tử trong mảng và kích thước mỗi phần tử là bao nhiêu byte.

Đối số cuối cùng là con trỏ hàm tới một hàm so sánh, mà trong chính nó lấy hai con trỏ void làm đối số. Bằng cách yêu cầu người gọi cung cấp hàm này, qsort() có thể sắp xếp các phần tử của bất kỳ kiểu dữ liệu nào một cách hiệu quả.

Dưới đây là ví dụ về một hàm so sánh như vậy, được sử dụng để so sánh các số thực (float). Lưu ý rằng bất kỳ hàm so sánh nào được truyền vào qsort() cần có chữ ký kiểu này. Cách thức hàm trở nên đa hình là bằng cách chuyển đổi các con trỏ void sang các con trỏ của loại phần tử mà chúng ta muốn so sánh.

|  |
| --- |
| int compare\_floats(const void \**a*, const void \**b*)  {  float fa = \*((float \*)*a*);  float fb = \*((float \*)*b*);  if (fa < fb)  return -1;  if (fa > fb)  return 1;  return 0;  } |

Vì chúng ta biết rằng qsort sẽ sử dụng hàm này để so sánh các số thực, chúng ta đưa ra chuyển đổi con trỏ void trở lại con trỏ float trước khi giải tham chiếu chúng.

Bây giờ, việc sử dụng hàm qsort đa hình trên một mảng "array" có độ dài "len" rất đơn giản:

|  |
| --- |
| qsort(*array*, *len*, sizeof(array[0]), *compare\_floats*); |

## Toán tử địa chỉ

Đối với bất kỳ đối tượng nào (ví dụ: biến, mảng, union, struct, con trỏ hoặc hàm), toán tử địa chỉ một ngôi có thể được sử dụng để truy cập địa chỉ của đối tượng đó.

|  |
| --- |
| int i = 1;  int \*p = NULL; |

Sau đó, câu lệnh p = &i; sao chép địa chỉ của biến i vào con trỏ p.

Câu lệnh này có nghĩa là p trỏ tới i.

printf("%d\n", \*p); in ra 1, đó là giá trị của biến i.

## Khởi tạo con trỏ

Khởi tạo con trỏ là một cách tốt để tránh sử dụng con trỏ vô lý. Quá trình khởi tạo đơn giản và không khác gì việc khởi tạo một biến bình thường.

|  |
| --- |
| #include <stddef.h>  int main()  {  int \*p1 = NULL;  char \*p2 = NULL;  float \*p3 = NULL;  /\* NULL là một macro được định nghĩa trong stddef.h, stdio.h, stdlib.h và string.h \*/  ...  } |

Trong hầu hết các hệ điều hành, việc vô tình sử dụng một con trỏ đã được khởi tạo thành NULL thường dẫn đến việc chương trình bị **crash** ngay lập tức, giúp dễ dàng xác định nguyên nhân của vấn đề. Sử dụng một con trỏ chưa được khởi tạo có thể dẫn đến các lỗi khó chẩn đoán.

**Cảnh báo:**

Kết quả của việc dereferencingmột con trỏ NULL là không xác định, do đó nó không nhất thiết phải gây ra lỗi crash ngay cả khi đó là hành vi tự nhiên của hệ điều hành mà chương trình đang chạy. Tối ưu hóa trình biên dịch có thể che giấu lỗi crash, làm cho lỗi xảy ra trước hoặc sau điểm trong mã nguồn mà giải tham chiếu con trỏ NULL, hoặc làm cho một số phần của mã chứa giải tham chiếu con trỏ NULL bị xóa bỏ một cách không mong đợi khỏi chương trình. Phiên bản dựng chế độ gỡ lỗi thường không thể thể hiện những hành vi này, nhưng điều này không được đảm bảo bởi tiêu chuẩn ngôn ngữ. Hành vi không mong đợi và/hoặc không mong muốn khác cũng được chấp nhận.

Vì NULL không bao giờ trỏ tới một biến, một vùng nhớ đã được cấp phát hoặc một hàm nào đó, nó an toàn khi sử dụng như một giá trị bảo vệ (guard value).

**Cảnh báo:**

Thông thường, NULL được định nghĩa là (void \*)0. Nhưng điều này không ám chỉ rằng địa chỉ bộ nhớ được gán là 0x0. Để có thêm thông tin chi tiết, hãy tham khảo C-faq về con trỏ NULL.

Lưu ý rằng bạn cũng có thể khởi tạo con trỏ để chứa các giá trị khác NULL.

|  |
| --- |
| int i1;  int main()  {  int \*p1 = &i1;  const char \*p2 = "A constant string to point to";  float \*p3 = malloc(10 \* sizeof(float));  } |

## Pointer to pointer

Trong C, một con trỏ có thể tham chiếu tới một con trỏ khác.

|  |
| --- |
| int main(void) {  int A = 42;  int\* pA = &A;  int\*\* ppA = &pA;  int\*\*\* pppA = &ppA;  printf("%d", \*\*\*pppA); /\* prints 42 \*/  return EXIT\_SUCCESS;  } |

Tuy nhiên, tham chiếu và tham chiếu trực tiếp không được phép.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void) {  int A = 42;  int\* pA = &A;  int\*\* ppA = &&A; /\* Compilation error here! \*/  int\*\*\* pppA = &&&A; /\* Compilation error here! \*/  ... |

## Con trỏ void\* như là đối số và giá trị trả về của các hàm tiêu chuẩn

Phiên bản > K&R

void\* là một kiểu chung cho các con trỏ đến các kiểu đối tượng. Một ví dụ về việc sử dụng con trỏ này là với hàm malloc, được khai báo như sau:

|  |
| --- |
| void\* malloc(size\_t); |

Kiểu trả về con trỏ void\* cho phép ta gán giá trị trả về của malloc cho một con trỏ đến bất kỳ kiểu đối tượng nào khác:

|  |
| --- |
| int\* vector = malloc(10 \* sizeof \*vector); |

Thường thì, ta nên tuân thủ việc không ép buộc chuyển đổi giá trị vào và ra từ con trỏ void. Trong trường hợp cụ thể của malloc(), điều này bởi vì nếu bạn quên bao gồm stdlib.h, trình biên dịch có thể giả định một kiểu trả về không đúng cho malloc() mà không cảnh báo. Điều này cũng liên quan đến việc sử dụng hành vi chính xác của con trỏ void\* để tuân thủ nguyên tắc DRY (don't repeat yourself); so sánh ví dụ trên với ví dụ sau, trong đó code sau chứa một số điểm không cần thiết khác nhau mà có thể gây ra lỗi nếu viết sai:

|  |
| --- |
| int\* vector = (int\*)malloc(10 \* sizeof int\*); |

Tương tự, các hàm như:

|  |
| --- |
| void\* memcpy(void \*restrict *target*, void const \*restrict *source*, *size\_t* *size*); |

có các đối số được chỉ định là void vì địa chỉ của bất kỳ đối tượng nào, bất kể kiểu, đều có thể được truyền vào. Ở đây cũng vậy, cuộc gọi không nên sử dụng ép kiểu:

|  |
| --- |
| unsigned char buffer[sizeof(int)];  int b = 67;  memcpy(buffer, &b, sizeof buffer); |

## Cùng một dấu hoa thị \* nhưng có ý nghĩa khác nhau

Một điều gây nhầm lẫn nhất khi làm việc với cú pháp con trỏ trong C và C++ là có thực sự hai ý nghĩa khác nhau mà áp dụng khi ký hiệu con trỏ, dấu hoa thị (\*) được sử dụng với một biến.

**Ví dụ:**

Trước tiên, ta sử dụng \* để khai báo một biến con trỏ.

|  |
| --- |
| int i = 5;  /\* 'p' là một con trỏ đến một số nguyên, được khởi tạo là NULL \*/  int \*p = NULL;  /\*'&i' cho ra địa chỉ của 'i', sau đó gán cho 'p' \*/  p = &i;  /\* 'p' bây giờ giữ địa chỉ của 'i' \*/ |

Khi bạn không khai báo (hoặc nhân), \* được sử dụng để dereferencemột biến con trỏ:

|  |
| --- |
| \*p = 123;  /\* 'p' trỏ đến 'i', vì vậy điều này thay đổi giá trị của 'i' thành 123 \*/ |

Khi bạn muốn một biến con trỏ hiện có giữ địa chỉ của biến khác, bạn không sử dụng \* mà hãy làm như sau:

|  |
| --- |
| p = &another\_variable; |

Một sự nhầm lẫn phổ biến giữa các newbie lập trình C xảy ra khi họ khai báo và khởi tạo một biến con trỏ cùng một lúc.

|  |
| --- |
| int \*p = &i; |

Bởi vì int i = 5; và int i; i = 5; cho kết quả giống nhau, một số người có thể nghĩ rằng int \*p = &i; và int \*p; \*p = &i; cũng cho kết quả giống nhau. Nhưng thực tế là không, int \*p; \*p = &i; sẽ cố gắng deference một con trỏ chưa được khởi tạo, dẫn đến undefined behavior (hành vi không xác định). Không bao giờ sử dụng \* khi bạn không khai báo hoặc giải tham chiếu một con trỏ.

**Kết luận:**

Dấu hoa thị (\*) có hai ý nghĩa riêng biệt trong C liên quan đến con trỏ, phụ thuộc vào nơi sử dụng nó. Khi sử dụng trong **khai báo biến**, giá trị ở phía bên phải của dấu bằng phải là một **giá trị con trỏ** trỏ đến một **địa chỉ** trong bộ nhớ. Khi sử dụng với một biến **đã khai báo**, dấu hoa thị sẽ **dereference** giá trị con trỏ, theo dõi đến vị trí đã trỏ tới trong bộ nhớ và cho phép gán hoặc truy xuất giá trị được lưu trữ tại đó.

Mang điều quan trọng

Quan trọng là chú ý khi làm việc với con trỏ. Hãy lưu ý khi bạn sử dụng dấu hoa thị và ý nghĩa khi sử dụng nó. Bỏ qua chi tiết nhỏ này có thể dẫn đến lỗi hoặc hành vi không xác định mà bạn thực sự không muốn phải đối mặt.

# SEQUENCE POINTS

## PHẦN 23.1 BIỂU THỨC KHÔNG CÓ THỨ TỰ

Phiên bản ≥ C11

Các biểu thức sau đây là không có thứ tự (unsequenced):

|  |
| --- |
| a + b;  a - b;  a \* b;  a / b;  a % b;  a & b;  a | b; |

Trong các ví dụ trên, biểu thức a có thể được đánh giá trước hoặc sau biểu thức b, b có thể được đánh giá trước a hoặc thậm chí chúng có thể được trộn lẫn nếu chúng tương ứng với một số hướng dẫn.

Một quy tắc tương tự cũng áp dụng cho việc gọi hàm:

|  |
| --- |
| f(a, b); |

Ở đây không chỉ a và b không có thứ tự (nghĩa là toán tử , trong lệnh gọi hàm không tạo ra "sequence point") mà còn cả f, biểu thức xác định hàm sẽ được gọi.

Tác động phụ có thể được áp dụng ngay sau khi tính giá trị hoặc hoãn lại cho đến thời điểm sau này.

Các biểu thức như:

|  |
| --- |
| x++ & x++;  f(x++, x++); /\* dấu ',' trong việc gọi hàm không phải là toán tử phẩy \*/  x++ \* x++;  a[i] = i++; |

Hoặc

|  |
| --- |
| x++ & x;  f(x++, x);  x++ \* x;  a[i++] = i; |

sẽ dẫn đến hành vi không xác định vì

* một sửa đổi của một đối tượng và bất kỳ truy cập nào khác đến nó phải được sắp xếp thứ tự
* thứ tự của việc tính toán và thứ tự mà các **side effect(1)** được áp dụng không được xác định.

(1) Mọi thay đổi về trạng thái của môi trường thực thi.

## PHẦN 23.2 BIỂU THỨC CÓ THỨ TỰ

Các biểu thức sau đây có thứ tự (sequenced):

|  |
| --- |
| a && b  a || b  a , b  a ? b : c  for ( a ; b ; c ) { ... } |

Trong tất cả các trường hợp, biểu thức a được tính toán đầy đủ và tất cả các tác động phụ được áp dụng trước khi b hoặc c được tính toán. Trong trường hợp thứ tư, chỉ một trong b hoặc c sẽ được tính toán. Trong trường hợp cuối cùng, b được tính toán đầy đủ và tất cả các tác động phụ được áp dụng trước khi c được tính toán.

Trong tất cả các trường hợp, việc tính toán biểu thức a được thực hiện theo thứ tự trước việc tính toán b hoặc c (hoặc ngược lại, việc tính toán b và c được thực hiện sau việc tính toán a).

Do đó, các biểu thức như

|  |
| --- |
| x++ && x++  x++ ? x++ : y++  (x = f()) && x != 0  for ( x = 0; x < 10; x++ ) { ... }  y = (x++, x++); |

đều có hành vi xác định.

## PHẦN 23.3 BIỂU THỨC ĐƯỢC XÁC ĐỊNH THỨ TỰ MỘT CÁCH KHÔNG XÁC ĐỊNH

Các lời gọi hàm như f(a) luôn áp dụng một **sequence point** giữa việc tính toán các đối số và đối tượng (ở đây là f và a) và việc gọi hàm thực tế. Nếu hai lời gọi hàm như vậy không có thứ tự, thì hai lời gọi hàm đó được xác định thứ tự một cách không xác định, tức là một lời gọi được thực hiện trước lời gọi khác và thứ tự này không được chỉ định.

|  |
| --- |
| unsigned counter = 0;  unsigned account(void) {  return counter++;  }  int main(void) {  printf("the order is %u %u\n", account(), account());  } |

Trong đoạn mã trên, việc sửa đổi counter trong quá trình tính giá trị đối số của printf là hợp lệ, chúng ta chỉ không biết lời gọi nào sẽ được thực hiện trước. Do thứ tự không được chỉ định, nó có thể thay đổi và không thể phụ thuộc vào. Vì vậy, kết quả có thể là:

|  |
| --- |
| the order is 0 1 |

Hoặc

|  |
| --- |
| the order is 1 0 |

Câu lệnh tương tự ở trên mà không cần gọi hàm trung gian

|  |
| --- |
| printf("the order is %u %u\n", counter++, counter++); // hành vi không xác định |

điều này có hành vi không xác định vì không có **sequence point** giữa hai việc sửa đổi của counter.

# CON TRỎ HÀM

Con trỏ hàm là các con trỏ **trỏ đến các hàm** thay vì các **kiểu dữ liệu**. Chúng có thể được sử dụng để cho phép tính linh hoạt trong việc **chọn hàm sẽ được gọi** tại thời điểm chạy.

## PHẦN 24.1 GIỚI THIỆU

Tương tự như char và int, hàm là một tính năng cơ bản của C. Do đó, bạn có thể khai báo một con trỏ trỏ đến hàm: điều này có nghĩa là bạn có thể chuyển vào một hàm nào đó để gọi vào một hàm khác để giúp nó hoàn thành công việc của nó. Ví dụ, nếu bạn có một hàm graph() để hiển thị một đồ thị, bạn có thể chuyển vào hàm nào sẽ được vẽ trong hàm graph().

|  |
| --- |
| extern unsigned int screenWidth;  extern void plotXY(double *x*, double *y*);  void graph(double *minX*, double *minY*,  double *maxX*, double *maxY*,  ???? \*fn) { // Xem cú pháp bên dưới  double stepX = (*maxX* - *minX*) / screenWidth;  for (double x = *minX*; x < *maxX*; x += stepX) {  double y = fn(x); // Lấy giá trị y cho x này bằng cách gọi hàm truyền vào fn()    if (*minY* <= y && y < *maxY*) {  plotXY(x, y); // Vẽ điểm tính toán được  }  }  } |

**Cách sử dụng:**

Mã trên sẽ vẽ đồ thị của bất kỳ hàm nào mà bạn chuyển vào - miễn là hàm đó đáp ứng một số tiêu chí nhất định: nghĩa là bạn chuyển vào một giá trị double và nhận lại một giá trị double. Có rất nhiều hàm như vậy - sin(), cos(), tan(), exp() v.v... - nhưng cũng có rất nhiều hàm không phải như vậy, như chính hàm graph() chẳng hạn!

**Cú pháp:**

Vậy làm sao bạn chỉ định các hàm nào bạn có thể chuyển vào trong hàm graph() và các hàm nào bạn không thể? Cách thông thường là sử dụng một cú pháp có thể không dễ đọc hoặc khó hiểu**:**

|  |
| --- |
| double (\*fn)(double); // fn là con trỏ hàm nhận một giá trị double và trả về một giá trị double |

Vấn đề ở trên là có hai thứ cố gắng được định nghĩa cùng một lúc: cấu trúc của hàm và việc nó là một con trỏ. Vậy nên, chia hai định nghĩa này ra thành hai! Nhưng bằng cách sử dụng **typedef**, bạn có thể đạt được một cú pháp tốt hơn (dễ đọc và hiểu hơn).

## PHẦN 24.2 TRẢ VỀ CON TRỎ HÀM TỪ MỘT HÀM

|  |
| --- |
| #include <stdio.h>  enum *Op*  {  ADD = '+',  SUB = '-',  };  // Hàm add: cộng a và b, trả về kết quả  int add(int *a*, int *b*)  {  return *a* + *b*;  }  // Hàm sub: trừ b từ a, trả về kết quả  int sub(int *a*, int *b*)  {  return *a* - *b*;  }  // Hàm getmath: trả về hàm toán học phù hợp  int (\*getmath(enum *Op* *op*))(int, int)  {  switch (*op*)  {  case ADD:  return &add;  case SUB:  return &sub;  default:  return NULL;  }  }  int main(void)  {  int a, b, c;  int (\*fp)(int, int);    fp = getmath(SUB);  a = 1, b = 2;  c = (\*fp)(a, b);  printf("%d - %d = %d\n", a, b, c);  return 0;  } |

## PHẦN 24.3 THỰC HÀNH TỐT NHẤT

**Sử dụng typedef**

Trong việc khai báo con trỏ hàm, nên sử dụng **typedef** thay vì khai báo con trỏ hàm một cách thủ công.

Cú pháp khai báo **typedef** cho con trỏ hàm như sau**:**

|  |
| --- |
| typedef *returnType* (\**name*) (parameters); |

**Ví dụ:**

Giả sử chúng ta có một hàm ***sort***, chấp nhận một con trỏ hàm ***compare*** như một đối số để so sánh hai phần tử.

Hàm so sánh ***compare*** cần trả về 0 nếu hai phần tử được coi là bằng nhau, giá trị dương nếu phần tử đầu tiên lớn hơn phần tử sau, và ngược lại trả về giá trị âm (nghĩa là phần tử đầu tiên nhỏ hơn phần tử sau).

Nếu không sử dụng **typedef**, chúng ta sẽ truyền một con trỏ hàm như một đối số vào hàm theo cách sau:

|  |
| --- |
| void sort(int (\**compare*)(const void \*elem1, const void \*elem2)) {  /\* bên trong khối này, hàm có tên là "compare" \*/  } |

Sử dụng **typedef**, chúng ta có thể viết như sau:

|  |
| --- |
| typedef int (\**compare\_func*)(const void \*, const void \*); |

Và sau đó, chúng ta có thể thay đổi function signature của sort như sau:

|  |
| --- |
| void sort(*compare\_func* *func*) {  /\* bên trong khối này, hàm có tên là "func" \*/  } |

Cả hai định nghĩa của sort đều chấp nhận bất kỳ hàm nào có dạng:

|  |
| --- |
| int compare(const void \**arg1*, const void \**arg2*) {  /\* Lưu ý rằng tên biến không cần phải là "elem1" và "elem2" \*/  } |

Con trỏ hàm là nơi duy nhất bạn nên bao gồm tính chất con trỏ của kiểu, ví dụ: không cố gắng định nghĩa các kiểu như typedef struct something\_struct \*something\_type. Điều này áp dụng ngay cả đối với một cấu trúc có thành viên mà không được truy cập trực tiếp bởi người gọi API, ví dụ: kiểu FILE trong stdio.h (mà bạn hiện đã nhận thấy không phải là con trỏ).

**Sử dụng con trỏ context:**

Một con trỏ hàm thường nên nhận một con trỏ void \* do người dùng cung cấp làm con trỏ context.

|  |
| --- |
| /\* hàm minimiser, chi tiết không quan trọng \*/  double findminimum(double (\**fptr*)(double x, double y, void \*ctx), void \**ctx*)  {  ...  /\* lặp đi lặp lại gọi như này \*/  temp = (\*fptr)(testx, testy, ctx);  }  /\* hàm chúng ta đang tối thiểu hóa, cộng hai hàm bậc ba \*/  double \*cubics(double *x*, double *y*, void \**ctx*)  {  double \*coeffsx = *ctx*;  double \*coeffsy = coeffx + 4;  return coeffsx[0] \* *x* \* *x* \* *x* + coeffsx[1] \* *x* \* *x* + coeffsx[2] \* *x* + coeffsx[3] +  coeffsy[0] \* *y* \* *y* \* *y* + coeffsy[1] \* *y* \* *y* + coeffsy[2] \* *y* + coeffsy[3];  }  void caller()  {  /\* context, các hệ số của hàm bậc ba \*/  double coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8};  double min;  min = findminimum(cubics, coeffs);  } |

Sử dụng con trỏ context đồng nghĩa với việc các tham số bổ sung không cần được khắc phục vào hàm trỏ tới, hoặc yêu cầu sử dụng biến toàn cục.

Hàm thư viện qsort() không tuân theo quy tắc này, và bạn thường có thể sử dụng không có context cho các hàm so sánh đơn giản. Nhưng với bất cứ thứ gì phức tạp hơn, con trỏ context trở nên cần thiết.

Xem thêm:

Con trỏ hàm (Function Pointers).

## PHẦN 24.4 GÁN CON TRỎ HÀM

|  |
| --- |
| #include <stdio.h>  /\* increment: nhận một số, tăng giá trị của nó lên một và trả về kết quả \*/  int increment(int *i*)  {  printf("tang %d len 1\n", *i*);  return *i* + 1;  }  /\* decrement: nhận một số, giảm giá trị của nó xuống một và trả về kết quả \*/  int decrement(int *i*)  {  printf("giam %d xuong 1\n", *i*);  return *i* - 1;  }  int main(void)  {  int num = 0; /\* khai báo số để tăng \*/  int (\*fp)(int); /\* khai báo con trỏ hàm \*/  fp = &increment; /\* thiết lập con trỏ hàm trỏ đến hàm increment \*/  num = (\*fp)(num); /\* tăng num lên \*/  num = (\*fp)(num); /\* tăng num lên lần thứ hai \*/  fp = &decrement; /\* thiết lập con trỏ hàm trỏ đến hàm decrement \*/  num = (\*fp)(num); /\* giảm num xuống \*/  printf("num luc nay la: %d\n", num);  return 0;  } |

Kết quả

|  |
| --- |
| tang 0 len 1  tang 1 len 1  giam 2 xuong 1  num luc nay la: 1 |

## PHẦN 24.5 GHI NHỚ ĐỂ VIẾT CON TRỎ HÀM

Tất cả các hàm C trên thực tế đều là các con trỏ tới **một vị trí trong bộ nhớ** chương trình (**địa chỉ**) nơi một số mã tồn tại. Công dụng chính của con trỏ hàm là cung cấp "**gọi lại**" cho các hàm khác (hoặc để mô phỏng các lớp và đối tượng).

Cú pháp của một hàm, như được định nghĩa sâu hơn trên trang này là:

|  |
| --- |
| returnType (\**name*)(prameters) |

Một cách ghi nhớ để viết định nghĩa con trỏ hàm là quy trình sau:

1. Bắt đầu bằng cách viết một khai báo hàm bình thường: returnType name(parameters)
2. Bọc tên hàm bằng cú pháp con trỏ: returnType (\*name)(parameters)

## PHẦN 24.6 KHÁI NIỆM CƠ BẢN

Giống như bạn có thể có một con trỏ tới một int , char , float , array/string , struct , v.v. - bạn có thể có một con trỏ tới một hàm.

**Khai báo con trỏ** nhận **giá trị trả về của hàm**(là địa chỉ), **tên của hàm** và loại **đối số/tham số** mà hàm nhận được

Giả sử bạn khởi tạo và khai báo hàm sau:

|  |
| --- |
| int addInt(int *n*, int *m*){  return n + m;  } |

Bạn có thể khai báo và khởi tạo một con trỏ tới hàm này:

|  |
| --- |
| int (\*functionPtrAdd)(int, int) = addInt; // or &addInt - the & is optional  /\*Bởi vì bản chất tên của 1 hàm đã trả về địa chỉ của hàm đó\*/ |

Nếu bạn có 1 **hàm void** thì trông nó sẽ như thế này:

|  |
| --- |
| void Print(void){  printf("look ma' - no hands, only pointers!\n");  } |

Sau đó khai báo con trỏ tới nó sẽ là:

|  |
| --- |
| void (\*functionPtrPrint)(void) = Print; |

Bản thân việc truy cập hàm sẽ yêu cầu dereferencing con trỏ:

|  |
| --- |
| sum = (\*functionPtrAdd)(2, 3); //sẽ gán cho tổng bằng 5  (\*functionPtrPrint)(); //sẽ in văn bản trong hàm Print |

Như đã thấy trong các ví dụ nâng cao hơn trong tài liệu này, việc khai báo một con trỏ tới hàm có thể trở nên lộn xộn nếu hàm được truyền nhiều hơn một vài tham số. Nếu bạn có một vài gợi ý tới các hàm có "cấu trúc" giống hệt nhau (cùng loại giá trị trả về và cùng loại tham số) thì tốt nhất bạn nên sử dụng lệnh **typedef** để tiết kiệm cho bạn một số thao tác gõ và để làm cho mã rõ ràng hơn:

|  |
| --- |
| #include <stdio.h>  typedef int (\**ptrInt*)(int, int);  int Add(int *a*, int *b*){  return *a* + *b*;  }  int Multiply(int *a*, int *b*){  return *a* \* *b*;  }  int main(){  *ptrInt* ptr1 = Add;  *ptrInt* ptr2 = Multiply;  printf("%d \n", (\*ptr1)(2,3)); //sẽ in ra 5  printf("%d \n", (\*ptr2)(2,3)); //sẽ in ra 6  return 0;  } |

Bạn cũng có thể tạo một **Mảng con trỏ hàm**. Nếu tất cả các con trỏ có cùng "cấu trúc":

|  |
| --- |
| int (\*array[2]) (int x, int y); // can hold 2 function pointers  array[0] = Add;  array[1] = Multiply; |

Cũng có thể định nghĩa một mảng các con trỏ hàm thuộc các loại khác nhau, mặc dù điều đó sẽ yêu cầu truyền bất cứ khi nào bạn muốn truy cập vào hàm cụ thể.

# THAM SỐ CỦA HÀM

## PHẦN 25.1 THAM SỐ ĐƯỢC TRUYỀN BẰNG GIÁ TRỊ (PASS BY VALUE)

Trong ngôn ngữ lập trình C, tất cả các tham số của hàm được truyền bằng giá trị, điều này đồng nghĩa rằng việc thay đổi giá trị của tham số trong hàm con sẽ không ảnh hưởng đến các biến cục bộ của hàm gọi.

|  |
| --- |
| #include <stdio.h>  void modify(int *v*) {  printf("modify 1: %d\n", *v*); /\* In kết quả: 0 \*/  *v* = 42;  printf("modify 2: %d\n", *v*); /\* In kết quả: 42 \*/  }  int main(void) {  int v = 0;  printf("main 1: %d\n", v); /\* In kết quả: 0 \*/  modify(v);  printf("main 2: %d\n", v); /\* In kết quả: 0, không phải 42 \*/  return 0;  } |

Bạn có thể sử dụng con trỏ để cho phép các hàm con thay đổi các biến cục bộ của các hàm gọi. Lưu ý rằng đây không phải là truyền theo tham chiếu (pass by reference) mà chỉ là giá trị con trỏ trỏ tới các biến cục bộ được truyền vào.

|  |
| --- |
| #include <stdio.h>  void modify(int\* *v*) {  printf("modify 1: %d\n", \**v*); /\* In kết quả: 0 \*/  \**v* = 42;  printf("modify 2: %d\n", \**v*); /\* In kết quả: 42 \*/  }  int main(void) {  int v = 0;  printf("main 1: %d\n", v); /\* In kết quả: 0 \*/  modify(&v);  printf("main 2: %d\n", v); /\* In kết quả: 42 \*/  return 0;  } |

Tuy nhiên, việc trả về địa chỉ của một biến cục bộ cho hàm gọi sẽ dẫn đến hành vi không xác định (undefined behavior). Xem Dereferencing a pointer to variable beyond its lifetime.

## PHẦN 25.2 TRUYỀN MẢNG VÀO HÀM

|  |
| --- |
| int getListOfFriends(size\_t *size*, int *friend\_indexes*[]) {  size\_t i = 0;  for (; i < *size*; i++) {  *friend\_indexes*[i] = i;  }  } |

Phiên bản C99 trở lên và phiên bản trước C11:

|  |
| --- |
| /\*Trong phiên bản C99 trở lên, việc sử dụng "void" và VLAs (Variable-Length Arrays)  trong tham số hàm ("int friend\_indexes[static size]")  yêu cầu C99 hoặc cao hơn. Trong C11, VLAs là tùy chọn.\*/  void getListOfFriends(size\_t *size*, int *friend\_indexes*[static *size*]) {  size\_t i = 0;  for (; i < *size*; i++) {  *friend\_indexes*[i] = 1;  } |

Ở đây, "static" bên trong dấu [] của tham số hàm yêu cầu rằng mảng đối số phải có ít nhất số phần tử được chỉ định (tức là "size" phần tử). Để có thể sử dụng tính năng này, chúng ta phải đảm bảo tham số "size" xuất hiện trước tham số mảng trong danh sách tham số.

Ta có thể sử dụng hàm "getListOfFriends()" như sau:

|  |
| --- |
| #define LIST\_SIZE (50)  int main(void) {  size\_t size\_of\_list = LIST\_SIZE;  int friends\_indexes[size\_of\_list];  getListOfFriends(size\_of\_list, friend\_indexes); /\* friend\_indexes decays to a pointer to the address of its 1st element: &friend\_indexes[0] \*/  /\* Here friend\_indexes carries: {0, 1, 2, ..., 49}; \*/  return 0;  } |

**Xem thêm:**

Truyền mảng nhiều chiều cho hàm

## PHẦN 25.3 THỨ TỰ THỰC THI THAM SỐ TRONG HÀM

Thứ tự thực hiện các tham số không được xác định trong lập trình C. Ở đây nó có thể thực hiện từ trái sang phải hoặc từ phải sang trái. Thứ tự phụ thuộc vào việc thực hiện.

|  |
| --- |
| #include <stdio.h>  void function(int *a*, int *b*)  {  printf("%d %d\n", *a*, *b*);  }  int main(void)  {  int a = 1;  function(a++, ++a);  return 0;  } |

## PHẦN 25.4 SỬ DỤNG THAM SỐ CON TRỎ ĐỂ TRẢ VỀ NHIỀU GIÁ TRỊ

Trong ngôn ngữ lập trình C, để dễ dàng mô phỏng việc trả về nhiều giá trị từ một hàm, một cách thông dụng là sử dụng con trỏ.

|  |
| --- |
| #include <stdio.h>  void Get(int\* *c*, double\* *d*) {  \**c* = 72;  \**d* = 175.0;  }  int main(void) {  int a = 0;  double b = 0.0;  Get(&a, &b);  printf("a: %d, b: %f\n", a, b);  return 0;  } |

## PHẦN 25.5 VÍ DỤ VỀ HÀM TRẢ VỀ CẤU TRÚC CHỨA CÁC GIÁ TRỊ CÓ MÃ LỖI

Hầu hết các ví dụ về hàm trả về giá trị liên quan đến việc cung cấp con trỏ làm một trong các đối số để cho phép hàm sửa đổi giá trị được trỏ tới, tương tự như sau. Giá trị trả về thực tế của hàm thường là một số kiểu dữ liệu, chẳng hạn như int để cho biết trạng thái của kết quả, cho dù nó có hoạt động hay không.

|  |
| --- |
| int func (int \**pIvalue*)  {  int iRetStatus = 0; /\* Default status is no change \*/  if (\**pIvalue* > 10) {  \**pIvalue* = \**pIvalue* \* 45; /\* Modify the value pointed to \*/  iRetStatus = 1; /\* indicate value was changed \*/  }    return iRetStatus; /\* Return an error code \*/  } |

Tuy nhiên, bạn cũng có thể sử dụng một struct làm giá trị trả về cho phép bạn trả về cả trạng thái lỗi cùng với các giá trị khác. Ví dụ.

|  |
| --- |
| typedef struct {  int iStat; /\* Mã lỗi trả về \*/  int iValue; /\* Giá trị trả về \*/  } *RetValue*;  *RetValue* func(int *iValue*) {  *RetValue* iRetStatus = {0, *iValue*}; /\* Mặc định là không có lỗi \*/    if (*iValue* > 10) {  iRetStatus.iValue = *iValue* \* 45; /\* Thay đổi giá trị trả về \*/  iRetStatus.iStat = 1; /\* Đánh dấu rằng giá trị đã được thay đổi \*/  }    return iRetStatus; /\* Trả về cấu trúc chứa mã lỗi và giá trị \*/  } |

Chức năng này sau đó có thể được sử dụng như ví dụ sau.

|  |
| --- |
| int usingFunc(int *iValue*) {  RetValue iRet = func(*iValue*);    if (iRet.iStat == 1) {  /\* Làm các việc với iRet.iValue, giá trị được trả về \*/  }    return 0;  } |

Hoặc nó có thể được sử dụng như sau.

|  |
| --- |
| int usingFunc(int *iValue*) {  RetValue iRet;  if ((iRet = func(*iValue*)).iStat == 1) {  /\* Làm các việc với iRet.iValue, giá trị được trả về \*/  }    return 0;  } |

# TRUYỀN MỘT MẢNG 2D ĐẾN MỘT HÀM

## PHẦN 26.1 TRUYỀN MỘT MẢNG 2D ĐẾN MỘT HÀM

Việc chuyển một mảng 2d cho một hàm có vẻ đơn giản và rõ ràng và chúng tôi vui vẻ viết:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define ROWS 3  #define COLS 2  void fun1(int \*\*, int, int);  int main()  {  int array\_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };  int n = ROWS;  int m = COLS;  fun1(array\_2D, n, m);  return EXIT\_SUCCESS;  }  void fun1(int \*\**a*, int *n*, int *m*)  {  int i, j;  for (i = 0; i < *n*; i++) {  for (j = 0; j < *m*; j++) {  printf("array[%d][%d]=%d\n", i, j, *a*[i][j]);  }  }  } |

Nhưng trình biên dịch, ở đây GCC trong phiên bản 4.9.4, không đánh giá cao nó.

|  |
| --- |
| $ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c11 passarr.c -o passarr  passarr.c: In function ‘main’:  passarr.c:16:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type  fun1(array\_2D, n, m);  ^  passarr.c:8:6: note: expected ‘int \*\*’ but argument is of type ‘int (\*)[2]’  void fun1(int \*\*, int, int); |

Có hai lý do cho điều này: vấn đề chính là mảng không phải là con trỏ và điều bất tiện thứ hai là cái gọi là **pointer decay**. Việc truyền một mảng cho một hàm sẽ **decay** mảng thành một con trỏ tới phần tử đầu tiên của mảng - trong trường hợp mảng 2D, nó phân rã thành một con trỏ tới hàng đầu tiên vì trong C, các mảng được sắp xếp theo hàng đầu tiên.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define ROWS 3  #define COLS 2  void fun1(int (\*)[COLS], int, int);  int main()  {  int array\_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };  int n = ROWS;  int m = COLS;  fun1(array\_2D, n, m);  return EXIT\_SUCCESS;  }  void fun1(int (\**a*)[COLS], int *n*, int *m*)  {  int i, j;  for (i = 0; i < *n*; i++) {  for (j = 0; j < *m*; j++) {  printf("array[%d][%d]=%d\n", i, j, *a*[i][j]);  }  }  } |

Cần phải vượt qua số lượng hàng, chúng không thể được tính toán.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #define ROWS 3  #define COLS 2  void fun1(int (\*)[COLS], int);  int main()  {  int array\_2D[ROWS][COLS] = { {1, 2}, {3, 4}, {5, 6} };  int rows = ROWS;  /\* works here because array\_2d is still in scope and still an array \*/  printf("MAIN: %zu\n",sizeof(array\_2D)/sizeof(array\_2D[0]));  fun1(array\_2D, rows);  return EXIT\_SUCCESS;  }  void fun1(int (\**a*)[COLS], int *rows*)  {  int i, j;  int n, m;  n = *rows*;  /\* Works, because that information is passed (as "COLS").  It is also redundant because that value is known at compile time (in "COLS"). \*/  m = (int) (sizeof(*a*[0])/sizeof(*a*[0][0]));    /\* Does not work here because the "decay" in "pointer decay" is meant  literally--information is lost. \*/  printf("FUN1: %zu\n",sizeof(*a*)/sizeof(*a*[0]));  for (i = 0; i < n; i++) {  for (j = 0; j < m; j++) {  printf("array[%d][%d]=%d\n", i, j, *a*[i][j]);  }  }  } |

Số lượng cột được xác định trước và do đó cố định tại thời điểm biên dịch, nhưng tiêu chuẩn tiền thân của tiêu chuẩn C hiện tại (đó là ISO/IEC 9899:1999, hiện tại là ISO/IEC 9899:2011) đã triển khai VLA (TODO: liên kết nó) và mặc dù tiêu chuẩn hiện tại làm cho nó trở thành tùy chọn, nhưng hầu hết tất cả các trình biên dịch C hiện đại đều hỗ trợ nó (CẦN LÀM: kiểm tra xem MS Visual Studio có hỗ trợ nó ngay bây giờ không).

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  /\* ALL CHECKS OMMITTED!\*/  void fun1(int (\*)[], int *rows*, int *cols*);  int main(int *argc*, char \*\**argv*)  {  int rows, cols, i, j;  if(*argc* != 3){  fprintf(stderr,"Usage: %s rows cols\n",*argv*[0]);  exit(EXIT\_FAILURE);  }  rows = atoi(*argv*[1]);  cols = atoi(*argv*[2]);  int array\_2D[rows][cols];  for (i = 0; i < rows; i++) {  for (j = 0; j < cols; j++) {  array\_2D[i][j] = (i + 1) \* (j + 1);  printf("array[%d][%d]=%d\n", i, j, array\_2D[i][j]);  }  }  fun1(array\_2D, rows, cols);  exit(EXIT\_SUCCESS);  }  void fun1(int (\**a*)[], int *rows*, int *cols*)  {  int i, j;  int n, m;  n = *rows*;  /\* Does not work anymore, no sizes are specified anymore  m = (int) (sizeof(a[0])/sizeof(a[0][0])); \*/  m = *cols*;  for (i = 0; i < n; i++) {  for (j = 0; j < m; j++) {  printf("array[%d][%d]=%d\n", i, j, *a*[i][j]);  }  }  } |

Điều này không hoạt động, trình biên dịch phàn nàn:

|  |
| --- |
| $ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c99 passarr.c -o passarr  passarr.c: In function ‘fun1’:  passarr.c:168:7: error: invalid use of array with unspecified bounds  printf("array[%d][%d]=%d\n", i, j, a[i][j]); |

Sẽ rõ ràng hơn một chút nếu chúng ta cố tình tạo ra lỗi trong lời gọi hàm bằng cách thay đổi phần khai báo thành void fun1(int \*\*a, int rows, int cols). Điều đó khiến trình biên dịch phàn nàn theo một cách khác, nhưng không kém phần mơ hồ

|  |
| --- |
| $ gcc-4.9 -O3 -g3 -W -Wall -Wextra -std=c99 passarr.c -o passarr  passarr.c: In function ‘main’:  passarr.c:208:8: warning: passing argument 1 of ‘fun1’ from incompatible pointer type  fun1(array\_2D, rows, cols);  ^  passarr.c:185:6: note: expected ‘int \*\*’ but argument is of type ‘int (\*)[(sizetype)(cols)]’  void fun1(int \*\*, int rows, int cols); |

Chúng ta có thể phản ứng theo nhiều cách, một trong số đó là bỏ qua tất cả và thực hiện một số trò tung hứng con trỏ khó đọc:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  /\* ALL CHECKS OMMITTED!\*/  void fun1(int (\*)[], int *rows*, int *cols*);  int main(int *argc*, char \*\**argv*)  {  int rows, cols, i, j;  if(*argc* != 3){  fprintf(stderr,"Usage: %s rows cols\n",*argv*[0]);  exit(EXIT\_FAILURE);  }  rows = atoi(*argv*[1]);  cols = atoi(*argv*[2]);  int array\_2D[rows][cols];  printf("Make array with %d rows and %d columns\n", rows, cols);  for (i = 0; i < rows; i++) {  for (j = 0; j < cols; j++) {  array\_2D[i][j] = i \* cols + j;  printf("array[%d][%d]=%d\n", i, j, array\_2D[i][j]);  }  }  fun1(array\_2D, rows, cols);  exit(EXIT\_SUCCESS);  }  void fun1(int (\**a*)[], int *rows*, int *cols*)  {  int i, j;  int n, m;  n = *rows*;  m = *cols*;  printf("\nPrint array with %d rows and %d columns\n", *rows*, *cols*);  for (i = 0; i < n; i++) {  for (j = 0; j < m; j++) {  printf("array[%d][%d]=%d\n", i, j, \*( (\**a*) + (i \* *cols* + j)));  }  }  } |

Hoặc chúng tôi làm đúng và chuyển thông tin cần thiết cho fun1. Để làm như vậy, chúng ta cần sắp xếp lại các đối số thành fun1: kích thước của cột phải xuất hiện trước khi khai báo mảng. Để giữ cho nó dễ đọc hơn, biến chứa số lượng hàng cũng đã thay đổi vị trí của nó và hiện là biến đầu tiên.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  /\* ALL CHECKS OMMITTED!\*/  void fun1(int *rows*, int *cols*, int (\*)[]);  int main(int *argc*, char \*\**argv*)  {  int rows, cols, i, j;  if(*argc* != 3){  fprintf(stderr,"Usage: %s rows cols\n",*argv*[0]);  exit(EXIT\_FAILURE);  }  rows = atoi(*argv*[1]);  cols = atoi(*argv*[2]);  int array\_2D[rows][cols];  printf("Make array with %d rows and %d columns\n", rows, cols);  for (i = 0; i < rows; i++) {  for (j = 0; j < cols; j++) {  array\_2D[i][j] = i \* cols + j;  printf("array[%d][%d]=%d\n", i, j, array\_2D[i][j]);  }  }  fun1(rows, cols, array\_2D);  exit(EXIT\_SUCCESS);  }  void fun1(int *rows*, int *cols*, int (\**a*)[*cols*])  {  int i, j;  int n, m;  n = *rows*;  m = *cols*;  printf("\nPrint array with %d rows and %d columns\n", *rows*, *cols*);  for (i = 0; i < n; i++) {  for (j = 0; j < m; j++) {  printf("array[%d][%d]=%d\n", i, j, *a*[i][j]);  }  }  } |

Điều này có vẻ khó xử đối với một số người, những người cho rằng thứ tự của các biến không quan trọng. Đó không phải là vấn đề lớn, chỉ cần khai báo một con trỏ và để nó trỏ đến mảng.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  /\* ALL CHECKS OMMITTED!\*/  void fun1(int *rows*, int *cols*, int \*\*);  int main(int *argc*, char \*\**argv*)  {  int rows, cols, i, j;  if(*argc* != 3){  fprintf(stderr,"Usage: %s rows cols\n",*argv*[0]);  exit(EXIT\_FAILURE);  }  rows = atoi(*argv*[1]);  cols = atoi(*argv*[2]);  int array\_2D[rows][cols];  printf("Make array with %d rows and %d columns\n", rows, cols);  for (i = 0; i < rows; i++) {  for (j = 0; j < cols; j++) {  array\_2D[i][j] = i \* cols + j;  printf("array[%d][%d]=%d\n", i, j, array\_2D[i][j]);  }  }  // a "rows" number of pointers to "int". Again a VLA  int \*a[rows];  // initialize them to point to the individual rows  for (i = 0; i < rows; i++) {  a[i] = array\_2D[i];  }  fun1(rows, cols, a);  exit(EXIT\_SUCCESS);  }  void fun1(int *rows*, int *cols*, int \*\**a*)  {  int i, j;  int n, m;  n = *rows*;  m = *cols*;  printf("\nPrint array with %d rows and %d columns\n", *rows*, *cols*);  for (i = 0; i < n; i++) {  for (j = 0; j < m; j++) {  printf("array[%d][%d]=%d\n", i, j, *a*[i][j]);  }  }  } |

## PHẦN 26.2 SỬ DỤNG MẢNG PHẲNG LÀM MẢNG 2D

Thường thì giải pháp đơn giản nhất chỉ đơn giản là chuyển các mảng 2D và cao hơn xung quanh dưới dạng bộ nhớ phẳng.

|  |
| --- |
| /\* create 2D array with dimensions determined at runtime \*/  double \*matrix = malloc(width \* height \* sizeof(double));  /\* initialise it (for the sake of illustration we want 1.0 on the diagonal) \*/  int x, y;  for (y = 0; y < height; y++)  {  for (x = 0; x < width; x++)  {  if (x == y)  matrix[y \* width + x] = 1.0;  else  matrix[y \* width + x] = 0.0;  }  }  /\* pass it to a subroutine \*/  manipulate\_matrix(matrix, width, height);  /\* do something with the matrix, e.g. scale by 2 \*/  void manipulate\_matrix(double \**matrix*, int *width*, int *height*)  {  int x, y;  for (y = 0; y < *height*; y++)  {  for (x = 0; x < *width*; x++)  {  *matrix*[y \* *width* + x] \*= 2.0;  }  }  } |

# XỬ LÝ LỖI

## PHẦN 27.1 ERRNO

Khi một hàm thư viện tiêu chuẩn bị lỗi, thường nó sẽ thiết lập errno thành mã lỗi phù hợp. Tiêu chuẩn C yêu cầu ít nhất 3 giá trị cho errno:

|  |  |
| --- | --- |
| Value | Meaning |
| EDOM | Domain error (Lỗi miền giá trị) |
| ERANGE | Range error (Lỗi phạm vi) |
| EILSEQ | Illegal multi-byte character sequence (Lỗi chuỗi ký tự không hợp lệ nhiều byte) |

## PHẦN 27.2 STRERROR

Nếu perror không đủ linh hoạt, bạn có thể lấy một mô tả lỗi có thể đọc được bởi người dùng bằng cách gọi strerror từ <string.h>.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  int main(int *argc*, char \**argv*[])  {  *FILE* \*fout;  int last\_error = 0;  if ((fout = fopen(*argv*[1], "w")) == NULL) {  last\_error = errno;  /\* reset errno and continue \*/  errno = 0;  }  /\* thực hiện một số xử lý và thử mở tệp theo cách khác, sau đó \*/  if (last\_error) {  fprintf(stderr, "fopen: Không thể mở %s để ghi: %s", *argv*[1], strerror(last\_error));  fputs("Chúc may mắn và tiếp tục", stderr);  }  /\* thực hiện một số xử lý khác \*/  return EXIT\_SUCCESS;  } |

## PHẦN 27.3 PERROR

Để in một thông báo lỗi có thể đọc được bởi người dùng vào stderr, gọi perror từ <stdio.h>.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int *argc*, char \**argv*[])  {  FILE \*fout;  if ((fout = fopen(argv[1], "w")) == NULL) {  perror("fopen: Không thể mở tệp để ghi");  return EXIT\_FAILURE;  }  return EXIT\_SUCCESS;  } |

# UNDEFINED BEHAVIOR (HÀNH VI KHÔNG XÁC ĐỊNH)

Trong ngôn ngữ lập trình C, một số biểu thức có thể cho kết quả không xác định (undefined behavior). Tiêu chuẩn của ngôn ngữ C cụ thể quyết định không định nghĩa cách mà một trình biên dịch phải hoạt động nếu nó gặp phải một biểu thức như vậy. Kết quả là, trình biên dịch có quyền tự do thực hiện bất cứ điều gì nó thấy phù hợp và có thể cho ra kết quả hữu ích, kết quả không mong đợi, hoặc thậm chí gây sự cố.

Mã lệnh sử dụng biểu thức gây ra hành vi không xác định có thể hoạt động đúng như mong muốn trên một hệ thống cụ thể với một trình biên dịch cụ thể, nhưng có thể không hoạt động đúng trên hệ thống khác, hoặc với một trình biên dịch khác, phiên bản trình biên dịch khác hoặc cấu hình trình biên dịch khác.

## DEREFERENCING MỘT CON TRỎ TỚI BIẾN NGOÀI THỜI GIAN TỒN TẠI CỦA NÓ

|  |
| --- |
| int\* foo(int *bar*)  {  int baz = 6;  baz += *bar*;  return &baz; /\* (&baz) copied to new memory location outside of foo. \*/  } /\* (1) The lifetime of baz and bar end here as they have automatic storage  \* duration (local variables), thus the returned pointer is not valid! \*/  int main (void)  {  int\* p;  p = foo(5); /\* (2) this expression's behavior is undefined \*/  \*p = \*p - 6; /\* (3) Undefined behaviour here \*/  return 0;  } |

Một số trình biên dịch chỉ ra điều này một cách hữu ích. Ví dụ: **gcc** cảnh báo với:

|  |
| --- |
| warning: function returns address of local variable [-Wreturn-local-addr] |

và **clang** cảnh báo với:

|  |
| --- |
| warning: address of stack memory associated with local variable 'baz' returned  [-Wreturn-stack-address] |

cho đoạn mã trên. Nhưng các trình biên dịch có thể không giúp được mã phức tạp.

1. Returning reference cho biến đã khai báo **static** là hành vi được xác định, vì biến không bị mất đi sau khi rời khỏi hàm hiện tại.
2. Theo ISO/IEC 9899:2011 6.2.4 §2, "Giá trị của một con trỏ trở nên không xác định được khi đối tượng mà nó trỏ tới hết thời gian tồn tại."
3. Dereferencing con trỏ được hàm trả về foo là hành vi không xác định vì bộ nhớ mà nó tham chiếu chứa một **giá trị không xác định**

## SAO CHÉP BỘ NHỚ CHỒNG CHÉO

Một loạt các chức năng thư viện tiêu chuẩn có tác dụng sao chép các chuỗi byte từ vùng bộ nhớ này sang vùng bộ nhớ khác. Hầu hết các chức năng này có hành vi không xác định khi vùng nguồn và đích trùng nhau.

Ví dụ về điều này:

|  |
| --- |
| #include <string.h> /\* for memcpy() \*/  char str[19] = "This is an example";  memcpy(str + 7, str, 10); |

... cố gắng sao chép 10 byte trong đó vùng bộ nhớ nguồn và đích chồng lên ba byte. Để hình dung:

|  |
| --- |
| overlapping area  |  \_ \_  | |  v v  T h i s i s a n e x a m p l e \0  ^ ^  | |  | destination  |  source |

Do chồng chéo, hành vi kết quả không được xác định.

Trong số các chức năng thư viện tiêu chuẩn có giới hạn loại này là **memcpy(), strcpy(), strcat(), sprintf()và sscanf()**. Tiêu chuẩn nói về những điều này và một số chức năng khác:

Nếu việc sao chép diễn ra giữa các đối tượng chồng lên nhau, thì hành vi đó không được xác định.

Chức năng này **memmove()**là ngoại lệ chính cho quy tắc này. Định nghĩa của nó xác định rằng hàm hoạt động như thể dữ liệu nguồn trước tiên được sao chép vào bộ đệm tạm thời và sau đó được ghi vào địa chỉ đích. Không có ngoại lệ đối với các vùng nguồn và đích chồng chéo, cũng như không cần có một vùng nào, vì vậy **memmove()**có hành vi được xác định rõ trong những trường hợp như vậy.

Sự khác biệt phản ánh hiệu quả so với . sự đánh đổi tổng thể. Việc sao chép như các chức năng này thực hiện thường xảy ra giữa các vùng bộ nhớ rời rạc và thường có thể biết tại thời điểm phát triển liệu một trường hợp sao chép bộ nhớ cụ thể có thuộc danh mục đó hay không. Giả định không chồng chéo tạo ra các triển khai tương đối hiệu quả hơn mà không tạo ra kết quả chính xác một cách đáng tin cậy khi giả định không đúng. Hầu hết các chức năng thư viện C được phép triển khai hiệu quả hơn và **memmove()**lấp đầy các khoảng trống, phục vụ các trường hợp nguồn và đích có thể hoặc chồng chéo. Tuy nhiên, để tạo ra hiệu ứng chính xác trong mọi trường hợp, nó phải thực hiện các thử nghiệm bổ sung và/hoặc sử dụng một triển khai tương đối kém hiệu quả hơn.

## SIGNED INTEGER OVERFLOW (TRÀN SỐ NGUYÊN CÓ DẤU)

Theo đoạn 6.5/5 của cả C99 và C11, việc đánh giá một biểu thức sẽ tạo ra hành vi không xác định nếu kết quả không phải là giá trị có thể biểu thị của loại biểu thức. Đối với các loại số học, đó được gọi là **overflow**(tràn). Unsigned integer( số nguyên không dấu) không tràn vì áp dụng đoạn 6.2.5/9, khiến bất kỳ kết quả unsigned(không dấu) nào nằm ngoài phạm vi sẽ bị giảm xuống giá trị trong phạm vi. Tuy nhiên, không có quy định tương tự cho các loại số nguyên đã ký ; những thứ này có thể và làm tràn, tạo ra hành vi không xác định. Ví dụ,

|  |
| --- |
| #include <limits.h> /\* to get INT\_MAX \*/  int main(void) {  int i = INT\_MAX + 1; /\* Overflow happens here \*/  return 0;  } |

Hầu hết các trường hợp của loại hành vi không xác định này đều khó nhận biết hoặc dự đoán hơn. Về nguyên tắc, tràn có thể phát sinh từ bất kỳ phép toán cộng, trừ hoặc nhân nào trên các số nguyên có dấu (tuân theo các chuyển đổi số học thông thường) khi không có giới hạn hiệu quả hoặc mối quan hệ giữa các toán hạng để ngăn chặn tràn. Ví dụ: chức năng này:

|  |
| --- |
| int square(int *x*) {  return *x* \* *x*; /\* overflows for some values of x \*/  } |

là hợp lý và nó thực hiện đúng đối với các giá trị đối số đủ nhỏ, nhưng hành vi của nó không được xác định đối với các giá trị đối số lớn hơn. Bạn không thể chỉ dựa vào chức năng để đánh giá liệu các chương trình gọi nó có thể hiện hành vi không xác định hay không. Nó phụ thuộc vào những đối số họ chuyển đến nó.

Mặt khác, hãy xem xét ví dụ đơn giản này về số học số nguyên có dấu an toàn chống tràn:

|  |
| --- |
| int zero(int *x*) {  return *x* - *x*; /\* Cannot overflow \*/  } |

Mối quan hệ giữa các toán hạng của toán tử trừ đảm bảo rằng phép trừ không bao giờ bị tràn. Hoặc xem xét ví dụ thực tế hơn một chút này:

|  |
| --- |
| int sizeDelta(FILE \**f1*, FILE \**f2*) {  int count1 = 0;  int count2 = 0;  while (fgetc(*f1*) != EOF) count1++; /\* might overflow \*/  while (fgetc(*f2*) != EOF) count2++; /\* might overflow \*/  return count1 - count2; /\* provided no UB to this point, will not overflow \*/  } |

Miễn là các bộ đếm không tràn riêng lẻ, các toán hạng của phép trừ cuối cùng sẽ không âm. Tất cả sự khác biệt giữa bất kỳ hai giá trị nào như vậy đều có thể biểu thị dưới dạng **int**.

## SỬ DỤNG MỘT BIẾN CHƯA ĐƯỢC KHỞI TẠO

|  |
| --- |
| int a;  printf("%d", a); |

Biến a là một int với thời lượng lưu trữ tự động. Đoạn mã ví dụ trên đang cố in giá trị của một biến chưa được khởi tạo (a chưa bao giờ được khởi tạo). Các biến tự động không được khởi tạo có giá trị không xác định; truy cập những thứ này có thể dẫn đến hành vi không xác định.

**Lưu ý:** Các biến có bộ nhớ cục bộ tĩnh(static) hoặc chuỗi, bao gồm các biến toàn cục không có từ khóa static, được khởi tạo thành 0 hoặc giá trị khởi tạo của chúng. Do đó, những điều sau đây là hợp pháp.

|  |
| --- |
| static int b;  printf("%d", b); |

Một lỗi rất phổ biến là không khởi tạo các biến đóng vai trò là bộ đếm thành 0. Bạn thêm giá trị cho chúng, nhưng vì giá trị ban đầu là rác nên bạn sẽ gọi Hành vi không xác định , chẳng hạn như trong câu hỏi **Compilation on terminal gives off pointer warning and strange symbols.**

**Ví dụ:**

|  |
| --- |
| #include <stdio.h>  int main(void) {  int i, counter;  for(i = 0; i < 10; ++i)  counter += i;  printf("%d\n", counter);  return 0;  } |

**Kết quả**

|  |
| --- |
| C02QT2UBFVH6-lm:~ gsamaras$ gcc main.c -Wall -o main  main.c:6:9: warning: variable 'counter' is uninitialized when used here [-Wuninitialized]  counter += i;  ^~~~~~~  main.c:4:19: note: initialize the variable 'counter' to silence this warning  int i, counter;  ^  = 0  1 warning generated.  C02QT2UBFVH6-lm:~ gsamaras$ ./main  32812 |

Các quy tắc trên cũng được áp dụng cho con trỏ. Ví dụ: các kết quả sau đây trong hành vi không xác định

|  |
| --- |
| int main(void)  {  int \*p;  p++; // Trying to increment an uninitialized pointer.  } |

Lưu ý rằng bản thân đoạn mã trên có thể không gây ra lỗi hoặc lỗi phân đoạn, nhưng việc cố gắng dereference con trỏ này sau đó sẽ gây ra hành vi không xác định.

## DATA RACE

## ĐỌC GIÁ TRỊ CỦA CON TRỎ ĐÃ ĐƯỢC GIẢI PHÓNG

Ngay cả khi chỉ đọc giá trị của một con trỏ đã được giải phóng (nghĩa là không cố gắng hủy đăng ký con trỏ) là hành vi không xác định (UB), VD

|  |
| --- |
| char \*p malloc(5);  free(*p*);  if(p= NULL) /\* NOTE: even without dereferencing, this may have UB \*/  {  } |

Trích dẫn ISO/IEC 9899:2011 , mục 6.2.4 §2:

[…] Giá trị của một con trỏ trở nên không xác định khi đối tượng mà nó trỏ tới (hoặc chỉ là quá khứ) sắp hết thời gian tồn tại.

Việc sử dụng bộ nhớ không xác định cho bất kỳ thứ gì, kể cả so sánh hoặc số học có vẻ vô hại, có thể có hành vi không xác định nếu giá trị có thể là đại diện bẫy cho các kiểu dữ liệu.

## SỬ DỤNG ĐỊNH DẠNG KHÔNG CHÍNH XÁC TRONG PRINTF

Sử dụng một mã định dạng không chính xác trong đối số đầu tiên để **printf** gọi hành vi không xác định. Ví dụ: mã bên dưới gọi mã định dạng 1 hành vi không xác định:

|  |
| --- |
| long z = 'B';;  printf("%c \n", z); |

Đây là một ví dụ khác:

|  |
| --- |
| printf("%f \n", z); |

Dòng mã trên là hành vi không xác định. %f biểu thị double. Tuy nhiên 0 là kiểu dữ liệu int.

Lưu ý rằng trình biên dịch của bạn thường có thể giúp bạn tránh những trường hợp như thế này, nếu bạn bật các cờ thích hợp trong quá trình biên dịch ( -Wformat trong clang và gcc). Từ ví dụ cuối cùng:

|  |
| --- |
| warning: format specifies type 'double' but the argument has type  'int' [-Wformat]  printf("%f\n",0);  ~~ ^  %d |

## SỬA ĐỔI CHUỖI KÝ TỰ

Trong ví dụ mã này, con trỏ **char p** được khởi tạo thành địa chỉ của một chuỗi ký tự. Cố gắng sửa đổi chuỗi ký tự có hành vi không xác định.

|  |
| --- |
| char \*p = "hello world";  p[0] = 'H'; // Undefined behavior |

Tuy nhiên, việc sửa đổi trong một mảng có thể thay đổi **char** trực tiếp hoặc thông qua một con trỏ đương nhiên không phải là hành vi không xác định, ngay cả khi bộ khởi tạo của nó là một chuỗi ký tự. Sau đây là đúng:

|  |
| --- |
| char a[] = "hello, world";  char \*p = a;  a[0] = 'H';  p[7] = 'W'; |

Đó là bởi vì chuỗi ký tự được sao chép một cách hiệu quả vào mảng mỗi khi mảng được khởi tạo (một lần đối với các biến có thời lượng tĩnh, mỗi lần mảng được tạo cho các biến có thời lượng luồng hoặc tự động — các biến có thời lượng được phân bổ không được khởi tạo) và nó là đúng để sửa đổi nội dung mảng.

## TRUYỀN MỘT CON TRỎ NULL ĐỂ CHUYỂN ĐỔI PRINTF(%S)

Việc %s chuyển đổi printf các trạng thái mà đối số tương ứng là một con trỏ tới phần tử ban đầu của một **mảng kiểu ký tự**. Một con trỏ **null** không trỏ đến phần tử ban đầu của bất kỳ mảng kiểu ký tự nào và do đó hành vi sau đây không được xác định:

|  |
| --- |
| char \*foo = NULL;  printf("%s", foo); /\* undefined behavior \*/ |

Tuy nhiên, hành vi không xác định không phải lúc nào cũng có nghĩa là chương trình gặp sự cố — một số hệ thống thực hiện các bước để tránh sự cố thường xảy ra khi con trỏ **null** bị dereferenced. Ví dụ **Glibc** sẽ in ra kết quả là:

|  |
| --- |
| (null) |

cho đoạn mã trên. Tuy nhiên, thêm (chỉ) một dòng mới vào chuỗi định dạng và bạn sẽ gặp sự cố:

|  |
| --- |
| char \*foo = 0;  printf("%s", foo); /\* undefined behavior \*/ |

Trong trường hợp này điều này xảy ra do GCC có một tối ưu hóa chuyển printf("%s\n", argument);thành lệnh gọi tới puts với puts(argument)và puts trong Glibc không xử lý các con trỏ **null**. Tất cả hành vi này là tiêu chuẩn phù hợp.

Lưu ý rằng con trỏ null khác với một chuỗi rỗng . Vì vậy, những điều sau đây là hợp lệ và không có hành vi không xác định. Nó sẽ chỉ in một dòng mới :

|  |
| --- |
| char \*foo = ' ';  printf("%s", foo); /\* undefined behavior \*/ |

## SỬA ĐỔI BẤT KÌ ĐỐI TƯỢNG NÀO NHIỀU LẦN GIỮA HAI SEQUENCE POINTS

|  |
| --- |
| int i = 42;  i = i++; /\* Assignment changes variable, post-increment as well \*/  int a = i++ + i--; |

Mã như thế này thường dẫn đến suy đoán về "giá trị kết quả" của i. Tuy nhiên, thay vì chỉ định một kết quả, các tiêu chuẩn C chỉ định rằng việc đánh giá một biểu thức như vậy sẽ tạo ra hành vi không xác định . Trước C2011, tiêu chuẩn đã chính thức hóa các quy tắc này theo cái gọi là các điểm trình tự :

Giữa điểm thứ tự trước đó và tiếp theo, một đối tượng vô hướng sẽ có giá trị được lưu trữ của nó được sửa đổi nhiều nhất một lần bằng cách đánh giá một biểu thức. Hơn nữa, giá trị trước sẽ chỉ được đọc để xác định giá trị được lưu trữ.

(Tiêu chuẩn C99, mục 6.5, khoản 2)

Sơ đồ đó tỏ ra hơi thô, dẫn đến một số biểu thức thể hiện hành vi không xác định đối với C99 mà lẽ ra không nên làm. C2011 giữ lại các điểm trình tự, nhưng giới thiệu một cách tiếp cận nhiều sắc thái hơn cho lĩnh vực này dựa trên trình tự và mối quan hệ mà nó gọi là "trình tự trước đó":

Nếu một tác dụng phụ trên một đối tượng vô hướng không được sắp xếp theo trình tự so với một tác dụng phụ khác trên cùng một đối tượng vô hướng hoặc một phép tính giá trị sử dụng giá trị của cùng một đối tượng vô hướng, thì hành vi đó sẽ không được xác định. Nếu có nhiều thứ tự cho phép của các biểu thức con của một biểu thức, hành vi sẽ không được xác định nếu tác dụng phụ không được sắp xếp theo trình tự như vậy xảy ra trong bất kỳ thứ tự nào.

(Chuẩn C2011, mục 6.5, khoản 2)

Các chi tiết đầy đủ của mối quan hệ "được giải trình tự trước" quá dài để mô tả ở đây, nhưng chúng bổ sung cho các điểm trình tự hơn là thay thế chúng, vì vậy chúng có tác dụng xác định hành vi cho một số đánh giá mà hành vi của chúng trước đây không được xác định. Cụ thể, nếu có một điểm trình tự giữa hai đánh giá, thì điểm trước điểm trình tự là "xếp thứ tự trước" điểm sau.

Ví dụ sau có hành vi được xác định rõ:

|  |
| --- |
| int i = 42;  i = (i++, i+42); /\* The comma-operator creates a sequence point \*/ |

Hành vi dưới đây là không xác định

|  |
| --- |
| int i = 42;  printf("%d %d\n", i++, i++); /\* commas as separator of function arguments are not comma-operators \*/ |

Như với bất kỳ dạng hành vi không xác định nào, việc quan sát hành vi thực tế của việc đánh giá các biểu thức vi phạm các quy tắc sắp xếp thứ tự không mang tính thông tin, ngoại trừ theo nghĩa hồi cứu. Tiêu chuẩn ngôn ngữ không cung cấp cơ sở để mong đợi những quan sát như vậy có thể dự đoán ngay cả về hành vi trong tương lai của cùng một chương trình.

## GIẢI PHÓNG BỘ NHỚ HAI LẦN

Giải phóng bộ nhớ 2 lần là hành vi không xác định:

|  |
| --- |
| int \* x = malloc(sizeof(int));  \*x = 9;  free(*x*);  free(*x*); |

Trích dẫn từ tiêu chuẩn (7.20.3.2. Chức năng miễn phí của C99):

Mặt khác, nếu đối số không khớp với một con trỏ trước đó được trả về bởi hàm calloc, malloc hoặc realloc hoặc nếu không gian đã được giải phóng bằng lệnh gọi free hoặc realloc, thì hành vi đó sẽ không được xác định.

## DỊCH CHUYỂN BIT BẰNG CÁCH SỬ DỤNG SỐ ÂM HOẶC VƯỢT QUÁ CHIỀU RỘNG CỦA KIỂU DỮ LIỆU

Nếu giá trị đếm thay đổi là một **giá trị âm** thì cả thao tác dịch bit trái và dịch bit phải đều không xác định

|  |
| --- |
| int x = 5 << -3; /\* undefined \*/  int x = 5 >> -3; /\* undefined \*/ |

Nếu dịch trái được thực hiện trên một **giá trị âm**, nó không được xác định:

|  |
| --- |
| int x = -5 << 3; /\* undefined \*/ |

Nếu dịch trái được thực hiện trên một giá trị dương và kết quả của giá trị toán học không thể biểu diễn trong kiểu dữ liệu, nó không xác định1:

|  |
| --- |
| /\* Assuming an int is 32-bits wide, the value '5 \* 2^72' doesn't fit  \* in an int. So, this is undefined. \*/    int x = 5 << 72; |

Lưu ý rằng dịch chuyển phải trên một giá trị âm (ví dụ: -5 >> 3) **không** phải là không xác định mà được **xác định theo triển khai.**

1 Trích dẫn ISO/IEC 9899:201x , mục 6.5.7:

Nếu giá trị của toán hạng bên phải là âm hoặc lớn hơn hoặc bằng chiều rộng của toán hạng bên trái được thăng cấp, hành vi không được xác định.

## QUAY LẠI TỪ MỘT HÀM ĐƯỢC KHAI BÁO BẰNG BỘ XÁC ĐỊNH HÀM ‘\_NORETURN’ HOẶC ‘NORETURN’

C11

Trình xác định chức năng \_Noreturnđã được giới thiệu trong C11. Tiêu đề <stdnoreturn.h>cung cấp một macro noreturnmở rộng thành \_Noreturn. Vì vậy, sử dụng \_Noreturnhoặc noreturntừ <stdnoreturn.h>là tốt và tương đương.

Một chức năng được khai báo với \_Noreturn (hoặc noreturn) không được phép trả về trình gọi của nó. Nếu một chức năng như vậy quay trở lại trình gọi của nó, thì hành vi đó không được xác định.

Trong ví dụ sau, func()được khai báo với noreturntrình xác định nhưng nó trả về trình gọi của nó.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <stdnoreturn.h>  noreturn void func(void);  void func(void)  {  printf("In func()...\n");  } /\* Undefined behavior as func() returns \*/  int main(void)  {  func();  return 0; |

**Gcc** và **clang** đưa ra các cảnh báo cho chương trình trên:

|  |
| --- |
| $ gcc test.c  test.c: In function ‘func’:  test.c:9:1: warning: ‘noreturn’ function does return  }  ^  $ clang test.c  test.c:9:1: warning: function declared 'noreturn' should not return [-Winvalid-noreturn]  }  ^ |

Một ví dụ sử dụng **noreturn** có hành vi được xác định rõ:

|  |
| --- |
| include <stdio.h>  #include <stdlib.h>  #include <stdnoreturn.h>  noreturn void my\_exit(void);  /\* calls exit() and doesn't return to its caller. \*/  void my\_exit(void)  {  printf("Exiting...\n");  exit(0);  }  int main(void)  {  my\_exit();  return 0; |

## TRUY CẬP BỘ NHỚ NGOÀI VÙNG NHỚ ĐƯỢC PHÂN BỔ

Một con trỏ tới một phần bộ nhớ chứa ncác phần tử chỉ có thể được hủy đăng ký nếu nó nằm trong phạm vi memoryvà memory + (n - 1). Hủy bỏ hội nghị một con trỏ bên ngoài phạm vi đó dẫn đến hành vi không xác định. Như một ví dụ, hãy xem xét đoạn mã sau:

|  |
| --- |
| int array[3];  int \*beyond\_array = array + 3;  \*beyond\_array = 0; /\* Accesses memory that has not been allocated. \*/ |

Dòng thứ ba truy cập phần tử thứ 4 trong một mảng chỉ dài 3 phần tử, dẫn đến hành vi không xác định. Tương tự, hành vi của dòng thứ hai trong đoạn mã sau cũng không được xác định rõ:

|  |
| --- |
| int array[3];  array[3] = 0; |

Lưu ý rằng việc trỏ qua phần tử cuối cùng của một mảng không phải là hành vi không xác định ( beyond\_array = array + 3 được xác định rõ ở đây), mà là dereferencing ( \*beyond\_array là hành vi không xác định). Quy tắc này cũng áp dụng cho bộ nhớ được cấp phát động (chẳng hạn như bộ đệm được tạo thông qua malloc).

## THAY ĐỔI BIẾN CONST THÔNG QUA CON TRỎ

|  |
| --- |
| int main (void)  {  const int foo\_readonly = 10;  int \*foo\_ptr;  foo\_ptr = (int \*)&foo\_readonly; /\* (1) This casts away the const qualifier \*/  \*foo\_ptr = 20; /\* This is undefined behavior \*/  return 0;  } |

Trích dẫn ISO/IEC 9899:201x , mục 6.7.3 §2:

Nếu một nỗ lực được thực hiện để sửa đổi một đối tượng được xác định bằng loại đủ điều kiện const thông qua việc sử dụng một giá trị có loại không đủ điều kiện const, hành vi đó sẽ không được xác định. […]

(1) Trong GCC, điều này có thể đưa ra cảnh báo sau: warning: assignment discards ‘const’ qualifier from pointer target type [-Wdiscarded-qualifiers]

## ĐỌC MỘT ĐỐI TƯỢNG CHƯA KHỞI TẠO KHÔNG ĐƯỢC HỖ TRỢ BỞI BỘ NHỚ

C11

Đọc một đối tượng sẽ gây ra hành vi không xác định, nếu đối tượng là 1 :

* chưa khởi tạo
* được xác định với thời lượng lưu trữ tự động
* địa chỉ của nó không bao giờ được thực hiện

Biến a trong ví dụ dưới đây thỏa mãn tất cả các điều kiện đó:

|  |
| --- |
| void Function(void)  {  int a;  int b = a;  } |

1 (Trích dẫn từ: ISO:IEC 9899:201X 6.3.2.1 Giá trị, mảng và bộ chỉ định hàm 2)

Nếu giá trị chỉ định một đối tượng có thời lượng lưu trữ tự động có thể đã được khai báo với lớp lưu trữ thanh ghi (chưa bao giờ lấy địa chỉ của nó) và đối tượng đó chưa được khởi tạo (không được khai báo bằng trình khởi tạo và không có sự gán cho nó trước khi sử dụng), hành vi này không được xác định.

## PHÉP CỘNG HOẶC PHÉP TRỪ CON TRỎ KHÔNG ĐƯỢC GIỚI HẠN ĐÚNG CÁCH

Đoạn mã sau có hành vi không xác định

|  |
| --- |
| char buffer[6] = "hello";  char \*ptr1 = buffer - 1; /\* undefined behavior \*/  char \*ptr2 = buffer + 5; /\* OK, pointing to the '\0' inside the array \*/  char \*ptr3 = buffer + 6; /\* OK, pointing to just beyond \*/  char \*ptr4 = buffer + 7; /\* undefined behavior \*/ |

Theo C11, nếu phép cộng hoặc phép trừ một con trỏ vào hoặc vượt ra ngoài, một đối tượng mảng và một kiểu số nguyên tạo ra kết quả không trỏ vào hoặc vượt ra ngoài, cùng một đối tượng mảng, thì hành vi đó không được xác định (6.5.6 ).

Ngoài ra, đó là hành vi không được xác định một cách tự nhiên để dereference một con trỏ trỏ đến bên ngoài mảng:

|  |
| --- |
| char buffer[6] = "hello";  char \*ptr3 = buffer + 6; /\* OK, pointing to just beyond \*/  char value = \*ptr3; /\* undefined behavior \*/ |

## DEREFERENCING CON TRỎ NULL

Đây là một ví dụ về dereferencing hội thảo một con trỏ NULL, gây ra hành vi không xác định.

|  |
| --- |
| int \*pointer = NULL;  int value = \*pointer; /\* Dereferencing happens here \*/ |

Một NULLcon trỏ được đảm bảo theo tiêu chuẩn C để so sánh bất bình đẳng với bất kỳ con trỏ nào với một đối tượng hợp lệ và việc dereferencing nó gọi hành vi không xác định.

## SỬ DỤNG FFLUSH TRÊN LUỒNG ĐẦU VÀO

Các tiêu chuẩn POSIX và C nêu rõ rằng việc sử dụng fflushtrên luồng đầu vào là hành vi không xác định. Chỉ được fflushxác định cho các luồng đầu ra.

|  |
| --- |
| #include <stdio.h>  int main()  {  int i;  char input[4096];  scanf("%i", &i);  fflush(stdin); // <-- undefined behavior  gets(input);  return 0;  } |

Không có cách tiêu chuẩn nào để loại bỏ các ký tự chưa đọc khỏi luồng đầu vào. Mặt khác, một số triển khai sử dụng fflush để xóa stdin bộ đệm. Microsoft xác định hành vi của fflush luồng đầu vào: Nếu luồng đang mở để nhập, fflush hãy xóa nội dung của bộ đệm. Theo POSIX.1-2008, hành vi của fflushkhông được xác định trừ khi tệp đầu vào có thể tìm kiếm được.

Xem sử dụng fflush(stdin) để biết thêm chi tiết.

## LIÊN KẾT KHÔNG NHẤT QUÁN CỦA CÁC ĐỊNH DANH

|  |
| --- |
| extern int var;  static int var; /\* Undefined behaviour \*/ |

C11, §6.2.2, 7 nói:

Nếu, trong một đơn vị dịch thuật, cùng một mã định danh xuất hiện với cả liên kết bên trong và bên ngoài, thì hành vi đó là không xác định.

Lưu ý rằng nếu một khai báo trước của một mã định danh hiển thị thì nó sẽ có liên kết của khai báo trước đó. C11, §6.2.2, 4 cho phép:

Đối với một mã định danh được khai báo với mã định danh lớp lưu trữ bên ngoài trong phạm vi hiển thị khai báo trước của mã định danh đó,31) nếu khai báo trước đó chỉ định liên kết bên trong hoặc bên ngoài, thì liên kết của mã định danh ở khai báo sau giống như liên kết được chỉ định tại khai báo trước đó. Nếu không có khai báo trước nào hiển thị hoặc nếu khai báo trước không chỉ định liên kết, thì mã định danh có liên kết bên ngoài.

|  |
| --- |
| /\* 1. This is NOT undefined \*/  static int var;  extern int var;  /\* 2. This is NOT undefined \*/  static int var;  static int var;  /\* 3. This is NOT undefined \*/  extern int var;  extern int var; |

## THIẾU CÂU LỆNH RETURN TRONG CÂU LỆNH TRẢ VỀ GIÁ TRỊ

|  |
| --- |
| int foo(void) {  /\* do stuff \*/  /\* no return here \*/  }  int main(void) {  /\* Trying to use the (not) returned value causes UB \*/  int value = foo();  return 0;  } |

Khi một hàm được khai báo để trả về một giá trị thì nó phải làm như vậy trên mọi đường dẫn mã có thể đi qua nó. Hành vi không xác định xảy ra ngay khi người gọi (đang mong đợi giá trị trả về) cố gắng sử dụng giá trị trả về 1 .

Lưu ý rằng hành vi không xác định chỉ xảy ra nếu người gọi cố gắng sử dụng/truy cập giá trị từ hàm. Ví dụ,

|  |
| --- |
| int foo(void) {  /\* do stuff \*/  /\* no return here \*/  }  int main(void) {  /\* The value (not) returned from foo() is unused. So, this program  \* doesn't cause \*undefined behaviour\*. \*/  foo();  return 0; |

C99

Hàm main()là một ngoại lệ đối với quy tắc này ở chỗ nó có thể bị kết thúc mà không có câu lệnh trả về vì giá trị trả về giả định của 0 sẽ tự động được sử dụng trong trường hợp này 2 .

1 ( ISO/IEC 9899:201x , 6.9.1/12)

Nếu đạt đến } kết thúc một hàm và giá trị của lệnh gọi hàm được sử dụng bởi người gọi, thì hành vi đó không được xác định.

2 ( ISO/IEC 9899:201x , 5.1.2.2.3/1)

đến } kết thúc chức năng chính trả về giá trị 0

## CHIA CHO SỐ 0

|  |
| --- |
| int x = 0;  int b = (8 / x); /\* integer division \*/ |

Hoặc

|  |
| --- |
| double x = 0.0;  double y = 5.0 / x; /\* floating point division \*/ |

Hoặc

|  |
| --- |
| int x = 0;  int y = 5 % x; /\* modulo operation \*/ |

Đối với dòng thứ hai trong mỗi ví dụ, trong đó giá trị của toán hạng thứ hai (x) bằng 0, hành vi không được xác định.

Lưu ý rằng hầu hết các triển khai toán học dấu phẩy động sẽ tuân theo một tiêu chuẩn (ví dụ: IEEE 754), trong trường hợp đó, các hoạt động như chia cho 0 sẽ có kết quả nhất quán (ví dụ: INFINITY) mặc dù tiêu chuẩn C cho biết hoạt động này không được xác định.

## CHUYỂN ĐỔI GIỮA CÁC LOẠI CON TRỎ TẠO RA KẾT QUẢ ĐƯỢC CĂN CHỈNH KHÔNG CHÍNH XÁC

Những điều sau đây có thể có hành vi không xác định do căn chỉnh con trỏ không chính xác:

|  |
| --- |
| char \*memory\_block = calloc(sizeof(uint32\_t) + 1, 1);  uint32\_t \*intptr = (uint32\_t\*)(memory\_block + 1); /\* possible undefined behavior \*/  uint32\_t mvalue = \*intptr; |

Hành vi không xác định xảy ra khi con trỏ được chuyển đổi. Theo C11, nếu chuyển đổi giữa hai loại con trỏ tạo ra kết quả được căn chỉnh không chính xác (6.3.2.3), thì hành vi đó là không xác định. uint32\_t Ví dụ, ở đây có thể yêu cầu căn chỉnh 2 hoặc 4.

callocmặt khác, được yêu cầu trả về một con trỏ được căn chỉnh phù hợp cho bất kỳ loại đối tượng nào; do đó memory\_blockđược căn chỉnh chính xác để chứa an uint32\_ttrong phần ban đầu của nó. Sau đó, trên một hệ thống uint32\_tcó yêu cầu căn chỉnh 2 hoặc 4, memory\_block + 1sẽ là một địa chỉ lẻ và do đó không được căn chỉnh chính xác.

Quan sát rằng các yêu cầu tiêu chuẩn C đã thực hiện thao tác truyền không được xác định. Điều này được áp đặt bởi vì trên các nền tảng nơi địa chỉ được phân đoạn, địa chỉ byte memory\_block + 1 thậm chí có thể không có biểu diễn đúng dưới dạng con trỏ số nguyên.

Truyền char \*tới các con trỏ tới các loại khác mà không có bất kỳ mối quan tâm nào đối với các yêu cầu căn chỉnh đôi khi được sử dụng không chính xác để giải mã các cấu trúc được đóng gói như tiêu đề tệp hoặc gói mạng.

memcpy:

|  |
| --- |
| memcpy(&*mvalue*, memory\_block + 1, sizeof *mvalue*); |

Ở đây không có chuyển đổi con trỏ uint32\_t\*diễn ra và các byte được sao chép từng cái một.

Hoạt động sao chép này cho ví dụ của chúng tôi chỉ dẫn đến giá trị hợp lệ của mvaluevì:

* Chúng tôi đã sử dụng calloc, vì vậy các byte được khởi tạo đúng cách. Trong trường hợp của chúng tôi, tất cả các byte đều có giá trị 0, nhưng bất kỳ cách khởi tạo thích hợp nào khác sẽ làm được.
* uint32\_tlà một loại chiều rộng chính xác và không có bit đệm
* Bất kỳ mẫu bit tùy ý nào cũng là biểu diễn hợp lệ cho bất kỳ loại không dấu nào.

## SỬA ĐỔI CHUỖI ĐƯỢC TRẢ VỀ BỞI CÁC HÀM GETENV, STRERROR, VÀ SETLOCALE

Sửa đổi các chuỗi được trả về bởi các hàm tiêu chuẩn và getenv()không được xác định. Vì vậy, việc triển khai có thể sử dụng lưu trữ tĩnh cho các chuỗi này.strerror()setlocale()

Hàm getenv(), C11, §7.22.4.7, 4 , cho biết:

Hàm getenv trả về một con trỏ tới một chuỗi được liên kết với thành viên danh sách phù hợp. Chuỗi được trỏ tới sẽ không bị chương trình sửa đổi, nhưng có thể bị ghi đè bởi lệnh gọi hàm getenv tiếp theo.

Hàm strerror(), C11, §7.23.6.3, 4 cho biết:

Hàm strerror trả về một con trỏ tới chuỗi, nội dung của nó là đặc trưng cục bộ. Mảng được trỏ tới sẽ không bị chương trình sửa đổi, nhưng có thể bị ghi đè bởi lệnh gọi hàm strerror tiếp theo.

Hàm setlocale(), C11, §7.11.1.1, 8 nói:

Con trỏ tới chuỗi được hàm setlocale trả về sao cho cuộc gọi tiếp theo với giá trị chuỗi đó và danh mục được liên kết của nó sẽ khôi phục phần ngôn ngữ đó của chương trình. Chuỗi được trỏ tới sẽ không bị chương trình sửa đổi, nhưng có thể bị ghi đè bởi lệnh gọi tiếp theo tới hàm setlocale.

Tương tự, localeconv()hàm trả về một con trỏ struct lconvkhông được sửa đổi.

Hàm localeconv(), C11, §7.11.2.1, 8 cho biết:

Hàm localeconv trả về một con trỏ tới đối tượng được điền vào. Cấu trúc được trỏ tới bởi giá trị trả về sẽ không bị chương trình sửa đổi, nhưng có thể bị ghi đè bởi một lệnh gọi tiếp theo tới hàm localeconv.

# TẠO SỐ NGẪU NHIÊN

## TẠO SỐ NGẪU NHIÊN CƠ BẢN

Trong ngôn ngữ lập trình C, chúng ta có thể sử dụng hàm **rand()** để tạo ra một số nguyên giả ngẫu nhiên trong khoảng từ **0** đến **RAND\_MAX** (bao gồm cả 0 và RAND\_MAX).

Để khởi tạo trình tạo số ngẫu nhiên giả, chúng ta sử dụng hàm **srand(int)**. Mỗi khi ta khởi tạo **rand()** với cùng một giá trị, nó sẽ tạo ra cùng một chuỗi các số ngẫu nhiên. Do đó, trước khi gọi **rand(),** chúng ta chỉ nên khởi tạo bộ tạo số ngẫu nhiên một lần duy nhất. Không nên lặp đi lặp lại việc khởi tạo, hoặc khởi tạo lại mỗi khi muốn tạo một loạt số ngẫu nhiên mới.

Thường thì, chúng ta sử dụng kết quả của hàm **time(NULL)** làm giá trị hạt giống. Nếu chương trình của bạn yêu cầu một chuỗi số ngẫu nhiên xác định, bạn có thể sử dụng cùng một giá trị hạt giống mỗi lần bắt đầu chương trình. Thông thường, điều này không cần thiết cho mã chính, nhưng nó rất hữu ích trong quá trình gỡ lỗi để tái tạo các lỗi.

Nên nhớ luôn khởi tạo bộ tạo số ngẫu nhiên. Nếu không được khởi tạo, nó sẽ hoạt động như đã được khởi tạo với **srand(1)**.

Dưới đây là một ví dụ minh họa:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int main(void) {  int i;  srand(time(NULL));  i = rand();  printf("Random value between [0, %d]: %d\n", RAND\_MAX, i);  return 0;  } |

**Kết quả:**

|  |
| --- |
| Random value between [0, 32767]: 29654 |

**Ghi chú:**

Tiêu chuẩn C không đảm bảo chất lượng của chuỗi ngẫu nhiên được tạo ra bởi rand(). Trong quá khứ, một số phiên bản của **rand()** đã có vấn đề nghiêm trọng về phân phối và tính ngẫu nhiên của các số được tạo ra. Việc sử dụng **rand()** không được khuyến nghị cho nhu cầu tạo số ngẫu nhiên nghiêm túc, chẳng hạn như trong mật mã học.

## TRÌNH TẠO ĐỒNG DƯ ĐƯỢC HOÁN VỊ

Đây là một trình tạo số ngẫu nhiên độc lập không dựa vào **rand()**hoặc các chức năng thư viện tương tự.

Tại sao bạn muốn một điều như vậy? Có thể bạn không tin tưởng trình tạo số ngẫu nhiên dựng sẵn trên nền tảng của mình hoặc có thể bạn muốn một nguồn ngẫu nhiên có thể tái tạo độc lập với bất kỳ triển khai thư viện cụ thể nào.

Mã này là PCG32 từ pcg-random.org , một RNG hiện đại, nhanh, có mục đích chung với các thuộc tính thống kê xuất sắc. Nó không an toàn về mặt mật mã, vì vậy đừng sử dụng nó để mã hóa.

|  |
| --- |
| #include <stdint.h>  /\* \*Really\* minimal PCG32 code / (c) 2014 M.E. O'Neill / pcg-random.org  \* Licensed under Apache License 2.0 (NO WARRANTY, etc. see website) \*/  typedef struct { *uint64\_t* state; *uint64\_t* inc; } *pcg32\_random\_t*;  *uint32\_t* pcg32\_random\_r(*pcg32\_random\_t*\* *rng*) {  *uint64\_t* oldstate = *rng*->state;  /\* Advance internal state \*/  *rng*->state = oldstate \* 6364136223846793005ULL + (*rng*->inc | 1);  /\* Calculate output function (XSH RR), uses old state for max ILP \*/  *uint32\_t* xorshifted = ((oldstate >> 18u) ^ oldstate) >> 27u;  *uint32\_t* rot = oldstate >> 59u;  return (xorshifted >> rot) | (xorshifted << ((-rot) & 31));  }  void pcg32\_srandom\_r(*pcg32\_random\_t*\* *rng*, *uint64\_t* *initstate*, *uint64\_t* *initseq*) {  *rng*->state = 0U;  *rng*->inc = (*initseq* << 1u) | 1u;  pcg32\_random\_r(*rng*);  *rng*->state += *initstate*;  pcg32\_random\_r(*rng*);  } |

Và đây là cách gọi nó

|  |
| --- |
| #include <stdio.h>  int main(void) {  *pcg32\_random\_t* rng; /\* RNG state \*/  int i;  /\* Seed the RNG \*/  pcg32\_srandom\_r(&rng, 42u, 54u);  /\* Print some random 32-bit integers \*/  for (i = 0; i < 6; i++)  printf("0x%08x\n", pcg32\_random\_r(&rng));    return 0;  } |

## THẾ HỆ XORSHIFT

Một giải pháp thay thế tốt và dễ dàng cho **rand()**các thủ tục còn thiếu sót là **xorshift** , một lớp trình tạo số giả ngẫu nhiên do George Marsaglia phát hiện . Trình tạo xorshift là một trong những trình tạo số ngẫu nhiên không bảo mật bằng mật mã nhanh nhất. Thông tin thêm và các triển khai ví dụ khác có sẵn trên trang Wikipedia xorshift

**Thực hiện ví dụ:**

|  |
| --- |
| #include <stdint.h>  /\* These state variables must be initialised so that they are not all zero. \*/  *uint32\_t* w, x, y, z;  *uint32\_t* xorshift128(void)  {  *uint32\_t* t = x;  t ^= t << 11U;  t ^= t >> 8U;  x = y; y = z; z = w;  w ^= w >> 19U;  w ^= t;  return w;  } |

## HẠN CHẾ TẠO TRONG MỘT PHẠM VI NHẤT ĐỊNH

Thường khi tạo các số ngẫu nhiên, việc tạo các số nguyên trong một phạm vi cụ thể hoặc giá trị p (xác suất) nằm trong khoảng từ 0.0 đến 1.0 là hữu ích. Trong khi phép chia lấy dư (modulus) có thể được sử dụng để chuyển đổi giá trị hạt giống thành một số nguyên nhỏ, điều này sẽ sử dụng các bit thấp, thường dẫn đến chu kỳ ngắn, gây hiện tượng lệch phân phối nhỏ nếu N lớn so với RAND\_MAX.

Đoạn mã sau định nghĩa một macro để tạo một giá trị p trong khoảng từ 0.0 đến 1.0 - epsilon:

|  |
| --- |
| #define uniform() (rand() / (RAND\_MAX + 1.0)) |

Sau đó, ta có thể sử dụng nó để tạo một số nguyên ngẫu nhiên thuộc phạm vi từ 0 đến N - 1:

|  |
| --- |
| i = (int)(uniform() \* N) |

Tuy nhiên, có một lỗi kỹ thuật, đó là RAND\_MAX có thể lớn hơn giá trị mà một biến kiểu double có thể biểu diễn chính xác. Điều này có nghĩa là RAND\_MAX + 1.0 được đánh giá thành RAND\_MAX và đôi khi hàm sẽ trả về giá trị 1.0. Tuy nhiên, khả năng xảy ra điều này là thấp và không đáng kể.

Nếu bạn cần tính toán chính xác hơn, bạn có thể xem xét việc sử dụng một bộ tạo số ngẫu nhiên có chất lượng cao hơn, nhưng điều này không phải là vấn đề lớn nếu bạn chỉ cần một phân phối số ngẫu nhiên đơn giản và không chính xác.

# TIỀN XỬ LÝ VÀ MACRO

Tất cả các lệnh tiền xử lý đều bắt đầu với ký tự # (ký tự gạch chéo ngược). Một macro trong ngôn ngữ lập trình C là một lệnh tiền xử lý được định nghĩa bằng cách sử dụng chỉ thị tiền xử lý #define. Trong giai đoạn tiền xử lý, trình tiền xử lý C (là một phần của trình biên dịch C) đơn giản chỉ thay thế nội dung của macro bất cứ khi nào tên của nó xuất hiện.

## HEADER INCLUDE GUARDS

Hầu hết các tệp header đều nên tuân theo cách sử dụng include guard như sau:

**my-header-file.h**

|  |
| --- |
| #ifndef MY\_HEADER\_FILE\_H  #define MY\_HEADER\_FILE\_H  // Nội dung mã cho tệp header  #endif |

Điều này đảm bảo rằng khi bạn sử dụng #include "my-header-file.h" ở nhiều nơi, bạn sẽ không nhận được các khai báo trùng lặp của hàm, biến, v.v. Giả sử có cấu trúc như sau:

**header-1.h**

|  |
| --- |
| typedef struct {  // ...  } *MyStruct*;  int myFunction(*MyStruct* \**value*); |

**header-2.h**

|  |
| --- |
| #include "header-1.h"  int myFunction2(*MyStruct* \**value*); |

**main.c**

|  |
| --- |
| #include "header-1.h"  #include "header-2.h"  int main() {  // thực hiện một số thao tác  } |

Đoạn mã trên có một vấn đề nghiêm trọng: nội dung chi tiết của MyStruct được định nghĩa hai lần, điều này không được phép. Điều này dẫn đến lỗi biên dịch khó tìm kiếm, vì một tệp tiêu đề bao gồm một tệp khác. Nếu bạn thay thế bằng "đặt cờ bảo vệ tiêu đề" như sau:

**header-1.h**

|  |
| --- |
| #ifndef HEADER\_1\_H  #define HEADER\_1\_H  typedef struct {  // ...  } *MyStruct*;  int myFunction(*MyStruct* \**value*);  #endif |

**header-2.h**

|  |
| --- |
| #ifndef HEADER\_2\_H  #define HEADER\_2\_H  #include "header-1.h"  int myFunction2(*MyStruct* \**value*);  #endif |

**main.c**

|  |
| --- |
| #include "header-1.h"  #include "header-2.h"  int main() {  // thực hiện một số thao tác  } |

Khi đó, quá trình tiền xử lý sẽ trở thành:

|  |
| --- |
| #ifndef HEADER\_1\_H  #define HEADER\_1\_H  typedef struct {  // ...  } MyStruct;  int myFunction(MyStruct \**value*);  #endif  #ifndef HEADER\_2\_H  #define HEADER\_2\_H  #ifndef HEADER\_1\_H  #define HEADER\_1\_H  typedef struct {  // ...  } MyStruct;  int myFunction(MyStruct \**value*);  #endif  int myFunction2(MyStruct \**value*);  #endif  int main() {  // thực hiện một số thao tác  } |

Khi trình biên dịch đến lần thứ hai của "header-1.h", HEADER\_1\_H đã được định nghĩa từ lần bao gồm trước đó. Vì vậy, nó sẽ thu gọn thành:

|  |
| --- |
| #define HEADER\_1\_H  typedef struct {  // ...  } *MyStruct*;  int myFunction(*MyStruct* \**value*);  #define HEADER\_2\_H  int myFunction2(*MyStruct* \**value*);  int main() {  // thực hiện một số thao tác  } |

Và do đó, không có lỗi biên dịch nữa.

Lưu ý: Có nhiều quy ước khác nhau cho việc đặt tên "đặt cờ bảo vệ tiêu đề". Một số người thích đặt tên là HEADER\_2\_H\_, trong khi một số người bao gồm tên dự án như MY\_PROJECT\_HEADER\_2\_H. Quan trọng là đảm bảo rằng quy ước bạn tuân thủ giúp mỗi tệp trong dự án của bạn có một "đặt cờ bảo vệ tiêu đề" duy nhất.

Nếu chi tiết cấu trúc không được bao gồm trong tệp tiêu đề, kiểu được khai báo sẽ là một "kiểu chưa hoàn chỉnh" hoặc "kiểu không rõ ràng". Các kiểu này có thể hữu ích, giấu thông tin cài đặt khỏi người dùng các hàm. Ví dụ, trong thư viện C chuẩn, kiểu FILE thường được coi là một "kiểu không rõ ràng" (mặc dù thông thường nó không được che giấu để các macro triển khai các hàm I/O chuẩn có thể sử dụng các thông tin nội bộ của cấu trúc). Trong trường hợp này, header-1.h có thể chứa:

|  |
| --- |
| #ifndef HEADER\_1\_H  #define HEADER\_1\_H  typedef struct MyStruct MyStruct;  int myFunction(MyStruct \**value*);  #endif |

Lưu ý rằng cấu trúc phải có tên thẻ (ở đây là MyStruct - nằm trong không gian tên thẻ, riêng biệt với không gian tên định danh của tên typedef MyStruct), và phần { ... } được bỏ qua. Điều này có n

## SỬ DỤNG #IF 0 ĐỂ CHẶN MÃ TRONG CÁC PHẦN

Nếu bạn có một số đoạn mã mà bạn đang xem xét gỡ bỏ hoặc muốn tạm thời vô hiệu hóa, bạn có thể chú thích nó bằng một khối chú thích.

|  |
| --- |
| /\* Khối chú thích xung quanh toàn bộ hàm để ngăn việc sử dụng nó.  \* Mục đích thực sự của hàm này là gì?  int myUnusedFunction(void)  {  int i = 5;  return i;  }  \*/ |

Tuy nhiên, nếu mã nguồn mà bạn bao quanh bằng một khối chú thích có những khối chú thích kiểu khối khác trong mã nguồn, kết thúc \*/ của các khối chú thích đã tồn tại có thể làm cho khối chú thích mới của bạn không hợp lệ và gây ra vấn đề biên dịch.

|  |
| --- |
| /\* Khối chú thích xung quanh toàn bộ hàm để ngăn việc sử dụng nó.  \* Mục đích thực sự của hàm này là gì?  int myUnusedFunction(void)  {  int i = 5;  /\* Trả về 5 \*/  return i;  }  \*/ |

Trong ví dụ trước đó, hai dòng cuối cùng của hàm và \*/ cuối cùng được nhìn thấy bởi trình biên dịch, do đó sẽ gây lỗi biên dịch. Một phương pháp an toàn hơn là sử dụng chỉ thị #if 0 xung quanh mã bạn muốn chặn.

|  |
| --- |
| #if 0  /\* #if 0 được đánh giá là sai, vì vậy tất cả những gì ở đây và #endif sẽ bị loại bỏ bởi trình tiền xử lý. \*/  int myUnusedFunction(void)  {  int i = 5;  return i;  }  #endif |

Lợi ích của phương pháp này là khi bạn muốn tìm mã đã chặn, bạn chỉ cần tìm "#if 0" thay vì tìm kiếm trong tất cả các chú thích.

Lợi ích rất quan trọng khác là bạn có thể lồng chặn mã bằng #if 0. Điều này không thể thực hiện được với chú thích.

Một cách thay thế cho việc sử dụng #if 0 là sử dụng một tên mà không được định nghĩa bằng #define, nhưng mô tả rõ hơn vì sao mã đang bị chặn. Ví dụ, nếu có một hàm có vẻ như là mã chết vô dụng, bạn có thể sử dụng #if defined(POSSIBLE\_DEAD\_CODE) hoặc #if defined(FUTURE\_CODE\_REL\_020201) cho mã cần thiết khi các chức năng khác được đặt vào chỗ. Khi quay lại để gỡ bỏ hoặc kích hoạt mã đó, các phần mã này sẽ dễ dàng tìm thấy.

## MACROS CÓ HÌNH THỨC CỦA HÀM

Macros có hình thức của hàm (function-like macros) tương tự như hàm nội tuyến (inline functions), chúng hữu ích trong một số trường hợp, chẳng hạn như ghi log tạm thời trong quá trình gỡ lỗi:

|  |
| --- |
| #ifdef DEBUG  # define LOGFILENAME "/tmp/logfile.log"  # define LOG(*str*) do { \  FILE \*fp = fopen(LOGFILENAME, "a"); \  if (fp) { \  fprintf(fp, "%s:%d %s\n", \_\_FILE\_\_, \_\_LINE\_\_, \  /\* không in con trỏ null \*/ \  str ? str : "<null>"); \  fclose(fp); \  } \  else { \  perror("Mở '" LOGFILENAME "' thất bại"); \  } \  } while (0)  #else  /\* Biến nó thành NOOP nếu không định nghĩa DEBUG. \*/  # define LOG(*LINE*) (void)0  #endif |

Trong cả hai trường hợp (có hoặc không có DEBUG), cuộc gọi đều hoạt động giống như một hàm với kiểu trả về là void. Điều này đảm bảo rằng các điều kiện if/else sẽ được hiểu theo cách dự kiến.

Trong trường hợp DEBUG, điều này được thực hiện thông qua cấu trúc do { ... } while (0). Trong trường hợp còn lại, (void)0 là một câu lệnh không có tác động phụ và chỉ đơn giản bị bỏ qua.

Một cách thay thế cho phương pháp sau có thể là:

|  |
| --- |
| #define LOG(*LINE*) do { /\* không làm gì cả \*/ } while (0) |

điều này khiến nó đồng dạng cú pháp với cách đầu tiên trong tất cả các trường hợp.

Nếu bạn sử dụng GCC, bạn cũng có thể triển khai một macro có hình thức hàm trả về kết quả sử dụng một phần mở rộng không chuẩn của GNU - biểu thức câu lệnh (statement expressions). Ví dụ:

|  |
| --- |
| #include <stdio.h>  #define POW(*X*, *Y*) \  ({ \  int i, r = 1; \  for (i = 0; i < Y; ++i) \  r \*= X; \  r; \ // giá trị trả về là kết quả của phép tính cuối cùng  })  int main(void)  {  int result;  result = POW(2, 3);  printf("Kết quả: %d\n", result);  } |

## CHÈN CÁC TỆP NGUỒN

Các chỉ thị tiền xử lý #include được sử dụng phổ biến như sau:

|  |
| --- |
| #include <stdio.h>  #include "myheader.h" |

Chỉ thị này sẽ thay thế câu lệnh bằng nội dung của tệp được trích dẫn. Dấu ngoặc nhọn (<>) được sử dụng cho các tệp tiêu đề được cài đặt trên hệ thống, trong khi dấu ngoặc kép ("") được sử dụng cho các tệp do người dùng cung cấp.

Các macros có thể mở rộng các macros khác một lần, như minh họa ví dụ sau:

|  |
| --- |
| #if VERSION == 1  #define INCFILE "vers1.h"  #elif VERSION == 2  #define INCFILE "vers2.h"  /\* và cứ tiếp tục như vậy \*/  #else  #define INCFILE "versN.h"  #endif  /\* ... \*/  #include INCFILE |

Trong ví dụ trên, khi gặp chỉ thị #include, trình tiền xử lý sẽ mở rộng INCFILE thành tên tệp tương ứng, dựa trên giá trị của macro VERSION. Điều này cho phép chọn tệp nguồn phù hợp tùy thuộc vào giá trị của macro.

## CONDITIONAL INCLUSION AND CONDITIONAL FUNCTION SIGNATURE MODIFICATION

Để chèn một khối mã theo điều kiện, trình tiền xử lý có một số chỉ thị (ví dụ như #if, #ifdef, #else, #endif, vv).

Ví dụ: Định nghĩa một macro printf có điều kiện, chỉ in ra khi macro DEBUG được định nghĩa:

|  |
| --- |
| #ifdef DEBUG  #define DLOG(*x*) (printf(x))  #else  #define DLOG(*x*)  #endif |

Các toán tử quan hệ thông thường của C có thể được sử dụng cho điều kiện #if:

|  |
| --- |
| #if \_\_STDC\_VERSION\_\_ >= 201112L  /\* Thực hiện điều gì đó cho C11 hoặc cao hơn \*/  #elif \_\_STDC\_VERSION\_\_ >= 199901L  /\* Thực hiện điều gì đó cho C99 \*/  #else  /\* Thực hiện điều gì đó cho phiên bản trước C99 \*/  #endif |

Các chỉ thị #if hoạt động tương tự như câu lệnh if của C, chúng chỉ chứa các biểu thức hằng số số nguyên và không chứa các ép kiểu. Nó hỗ trợ một toán tử một ngôi bổ sung, defined(identifier), trả về 1 nếu identifier được định nghĩa và 0 trong trường hợp ngược lại.

|  |
| --- |
| #if defined(DEBUG) && !defined(QUIET)  #define DLOG(*x*) (printf(x))  #else  #define DLOG(*x*)  #endif |

**Conditional Function Signature Modification**

Trong hầu hết các trường hợp, một phiên bản release của ứng dụng được kỳ vọng có ít đầu tư phụ nhất có thể. Tuy nhiên, trong quá trình kiểm tra một phiên bản trung gian, thông tin bổ sung về các vấn đề tìm thấy có thể hữu ích.

Ví dụ: giả sử có một số chức năng SHORT SerOpPluAllRead(PLUIF \*pPif, USHORT usLockHnd)mà khi thực hiện xây dựng thử nghiệm, nó mong muốn sẽ tạo nhật ký về việc sử dụng nó. Tuy nhiên, chức năng này được sử dụng ở nhiều nơi và mong muốn rằng khi tạo nhật ký, một phần thông tin là để biết chức năng được gọi từ đâu.

Vì vậy, bằng cách sử dụng trình biên dịch có điều kiện, bạn có thể có nội dung như sau trong tệp bao gồm khai báo hàm. Điều này thay thế phiên bản tiêu chuẩn của hàm bằng phiên bản gỡ lỗi của hàm. Bộ tiền xử lý được sử dụng để thay thế các lệnh gọi hàm SerOpPluAllRead()bằng các lệnh gọi hàm SerOpPluAllRead\_Debug()với hai đối số bổ sung, tên của tệp và số dòng nơi hàm được sử dụng.

Quá trình biên dịch có điều kiện được sử dụng để chọn có ghi đè chức năng tiêu chuẩn bằng phiên bản gỡ lỗi hay không.

|  |
| --- |
| #if 0  // Khai báo và nguyên mẫu của phiên bản gỡ lỗi của hàm.  SHORT SerOpPluAllRead\_Debug(PLUIF \*pPif, USHORT usLockHnd, char \*aszFilePath, int nLineNo);  // Định nghĩa macro để thay thế cuộc gọi hàm bằng cuộc gọi hàm SerOpPluAllRead\_Debug() với hai đối số bổ sung,  // tên tệp và số dòng nơi hàm được sử dụng.  #define SerOpPluAllRead(pPif,usLock) SerOpPluAllRead\_Debug(pPif,usLock,\_\_FILE\_\_,\_\_LINE\_\_)  #else  // Khai báo hàm tiêu chuẩn mà thông thường được sử dụng trong quá trình xây dựng.  SHORT SerOpPluAllRead(PLUIF \**pPif*, USHORT *usLockHnd*);  #endif |

Điều này cho phép bạn ghi đè phiên bản tiêu chuẩn của hàm SerOpPluAllRead() bằng một phiên bản sẽ cung cấp tên của tệp và số dòng trong tệp nơi hàm được gọi.

Lưu ý rằng bất kỳ tệp nào sử dụng hàm này đều phải bao gồm tệp tiêu đề mà trong đó phương pháp này được sử dụng để trình tiền xử lý có thể sửa đổi hàm. Nếu không, bạn sẽ nhìn thấy lỗi liên kết.

**Có một cân nhắc quan trọng**: bất kỳ tệp nào sử dụng chức năng này phải bao gồm tệp tiêu đề nơi phương pháp này được sử dụng để bộ tiền xử lý sửa đổi chức năng. Nếu không, bạn sẽ thấy một lỗi liên kết.

Định nghĩa của hàm sẽ giống như sau. Điều mà nguồn này làm là yêu cầu bộ tiền xử lý đổi tên hàm SerOpPluAllRead()thành SerOpPluAllRead\_Debug()và sửa đổi danh sách đối số để bao gồm hai đối số bổ sung, một con trỏ tới tên của tệp nơi hàm được gọi và số dòng trong tệp mà tại đó hàm được gọi. chức năng được sử dụng.

|  |
| --- |
| // Khai báo hàm `SerOpPluAllRead\_Special` để sử dụng sau này  SHORT SerOpPluAllRead\_Special(PLUIF \**pPif*, USHORT *usLockHnd*);  // Khai báo hàm `SerOpPluAllRead\_Debug` với thêm thông tin vị trí gọi hàm để tạo log  SHORT SerOpPluAllRead\_Debug(PLUIF \**pPif*, USHORT *usLockHnd*, char \**aszFilePath*, int *nLineNo*)  {  int iLen = 0;  char xBuffer[256];  // Chỉ in ra 30 ký tự cuối cùng của tên tệp để rút ngắn log  iLen = strlen(*aszFilePath*);  if (iLen > 30) {  iLen = iLen - 30;  } else {  iLen = 0;  }  // Tạo thông điệp log với thông tin về file và dòng gọi hàm  sprintf(xBuffer, "SerOpPluAllRead\_Debug(): husHandle = %d, File %s, lineno = %d",  *pPif*->husHandle, *aszFilePath* + iLen, *nLineNo*);  IssueDebugLog(xBuffer);  // Tiếp tục xử lý bằng cách gọi hàm `SerOpPluAllRead\_Special`  return SerOpPluAllRead\_Special(*pPif*, *usLockHnd*);  }  // Hàm `SerOpPluAllRead\_Special` là phiên bản thay thế khi muốn tạo log  SHORT SerOpPluAllRead\_Special(PLUIF \**pPif*, USHORT *usLockHnd*)  {  // ... (các xử lý gỡ lỗi khác, nếu cần)  // Tiếp tục xử lý chính của hàm, chẳng hạn như gọi hàm `OpPluAllRead`  if (STUB\_SELF == SstReadAsMaster()) {  return OpPluAllRead(*pPif*, *usLockHnd*);  }  // Hoặc thực hiện một số xử lý khác và trả về kết quả  return OP\_NOT\_MASTER;  }  // Hàm `SerOpPluAllRead` bình thường, sử dụng trong các phiên bản chính thức  SHORT SerOpPluAllRead(PLUIF \**pPif*, USHORT *usLockHnd*)  {  // ... (các xử lý chính của hàm)  // Hoặc gọi hàm `SerOpPluAllRead\_Special` hoặc `SerOpPluAllRead\_Debug` tùy thuộc vào việc DEBUG đã được định nghĩa hay không  #if defined(SerOpPluAllRead)  return SerOpPluAllRead\_Special(pPif, usLockHnd);  #else  return SerOpPluAllRead\_Debug(*pPif*, *usLockHnd*, \_\_FILE\_\_, \_\_LINE\_\_);  #endif  } |

## \_ \_ CPLUSPLUS ĐỂ SỬ DỤNG CÁC PHẦN TỬ C NGOÀI MÃ C ++ KHI BIÊN DỊCH C++ BẰNG TÊN BỊ MÃ HÓA

Đôi khi một tệp bao gồm phải tạo đầu ra khác với bộ tiền xử lý tùy thuộc vào việc trình biên dịch là trình biên dịch C hay trình biên dịch C++ do sự khác biệt về ngôn ngữ.

Ví dụ: một hàm hoặc bên ngoài khác được xác định trong tệp nguồn C nhưng được sử dụng trong tệp nguồn C++. Vì C++ sử dụng xáo trộn tên (hoặc trang trí tên) để tạo các tên hàm duy nhất dựa trên các loại đối số của hàm, nên một khai báo hàm C được sử dụng trong tệp nguồn C++ sẽ gây ra lỗi liên kết. Trình biên dịch C++ sẽ sửa đổi tên bên ngoài được chỉ định cho đầu ra của trình biên dịch bằng cách sử dụng các quy tắc xáo trộn tên cho C++. Kết quả là lỗi liên kết do không tìm thấy bên ngoài khi đầu ra của trình biên dịch C++ được liên kết với đầu ra của trình biên dịch C.

Vì trình biên dịch C không thực hiện xáo trộn tên nhưng trình biên dịch C++ thực hiện cho tất cả các nhãn bên ngoài (tên hàm hoặc tên biến) do trình biên dịch C++ tạo ra, một macro tiền xử lý được xác định trước, , đã được giới thiệu \_\_cplusplusđể cho phép phát hiện trình biên dịch.

Để khắc phục sự cố đầu ra trình biên dịch không tương thích này cho các tên bên ngoài giữa C và C++, macro \_\_cplusplusđược xác định trong Bộ tiền xử lý C++ và không được xác định trong Bộ tiền xử lý C. Tên macro này có thể được sử dụng với #ifdefchỉ thị tiền xử lý có điều kiện hoặc #ifvới defined()toán tử để cho biết mã nguồn hoặc tệp bao gồm đang được biên dịch thành C++ hay C.

|  |
| --- |
| #ifdef \_\_cplusplus  printf("C++\n");  #else  printf("C\n");  #endif |

Hoặc bạn có thể sử dụng

|  |
| --- |
| #if defined(\_\_cplusplus)  printf("C++\n");  #else  printf("C\n");  #endif |

Để chỉ định tên hàm chính xác của hàm từ tệp nguồn C được biên dịch bằng trình biên dịch C đang được sử dụng trong tệp nguồn C++, bạn có thể kiểm tra hằng số đã xác định để sử dụng \_\_cplusplus để extern "C" { /\* ... \*/ };khai báo các phần bên ngoài C khi tệp tiêu đề được bao gồm trong tệp nguồn C++. Tuy nhiên, khi được biên dịch bằng trình biên dịch C, thì extern "C" { \*/ ... \*/ };không được sử dụng. Quá trình biên dịch có điều kiện này là cần thiết vì extern "C" { /\* ... \*/ };hợp lệ trong C++ nhưng không hợp lệ trong C.

|  |
| --- |
| #ifdef \_\_cplusplus  // if we are being compiled with a C++ compiler then declare the  // following functions as C functions to prevent name mangling.  extern "C" {  #endif  // exported C function list.  int foo (void);  #ifdef \_\_cplusplus  // if this is a C++ compiler, we need to close off the extern declaration.  };  #endif |

## DÁN MÃ THÔNG BÁO

Dán mã thông báo cho phép một người dán hai đối số macro lại với nhau. Ví dụ, front##back sản lượng frontback. Một ví dụ nổi tiếng là tiêu đề của Win32 <TCHAR.H>. Trong chuẩn C, người ta có thể viết L"string"để khai báo một chuỗi ký tự rộng. Tuy nhiên, Windows API cho phép một người chuyển đổi giữa chuỗi ký tự rộng và chuỗi ký tự hẹp chỉ bằng cách #defineing UNICODE. Để triển khai chuỗi ký tự, TCHAR.H hãy sử dụng cái này

|  |
| --- |
| #ifdef UNICODE  #define TEXT(*x*) L##x  #endif |

Bất cứ khi nào người dùng viết TEXT( "hello, world") và UNICODE được xác định, bộ tiền xử lý C sẽ ghép nối L và đối số macro. L nối với "hello, world"cho L"hello, world".

## MACRO ĐƯỢC XÁC ĐỊNH TRƯỚC

Macro được xác định trước là macro đã được bộ xử lý trước, C hiểu mà không cần chương trình xác định nó. Những ví dụ bao gồm

### Macro xác định trước bắt buộc

* \_\_FILE\_\_ cung cấp tên tệp của tệp nguồn hiện tại (một chuỗi ký tự),
* \_\_LINE\_\_ đối với số dòng hiện tại (một hằng số số nguyên),
* \_\_DATE\_\_ cho ngày biên dịch (một chuỗi ký tự),
* \_\_TIME\_\_ cho thời gian biên dịch (một chuỗi ký tự).

Ngoài ra còn có một mã định danh được xác định trước có liên quan, \_\_func\_\_(ISO/IEC 9899:2011 §6.4.2.2), không phải là macro:

Mã định danh \_\_func\_\_sẽ được người dịch ngầm khai báo như thể, ngay sau dấu ngoặc mở của mỗi định nghĩa hàm, khai báo:

|  |
| --- |
| static const char \_\_func\_\_[] = "function-name"; |

đã xuất hiện, trong đó **function-name** là tên của hàm chứa từ vựng.

\_\_FILE\_\_và đặc biệt hữu ích cho mục đích gỡ lỗi \_\_LINE\_\_. \_\_func\_\_Ví dụ:

|  |
| --- |
| fprintf(*stderr*, "%s: %s: %d: Denominator is 0", \_\_FILE\_\_, \_\_func\_\_, \_\_LINE\_\_); |

Các trình biên dịch trước C99, có thể hỗ trợ hoặc không \_\_func\_\_hoặc có thể có một macro hoạt động giống như vậy nhưng được đặt tên khác. Ví dụ: gcc được sử dụng \_\_FUNCTION\_\_ở chế độ C89.

Các macro dưới đây cho phép yêu cầu chi tiết về việc triển khai:

* \_\_STDC\_VERSION\_\_Phiên bản của Tiêu chuẩn C được triển khai. Đây là một số nguyên không đổi sử dụng định dạng yyyymmL(giá trị 201112Lcho C11, giá trị 199901Lcho C99; giá trị này không được xác định cho C89/C90)
* \_\_STDC\_HOSTED\_\_ 1nếu đó là triển khai được lưu trữ, thì khác 0.
* \_\_STDC\_\_Nếu 1, việc triển khai tuân theo Tiêu chuẩn C.

### Macro được xác định trước khác (không bắt buộc)

ISO/IEC 9899:2011 §6.10.9.2 Macro môi trường:

* \_\_STDC\_ISO\_10646\_\_ Hằng số nguyên có dạng yyyymmL(ví dụ: 199712L). Nếu ký hiệu này được xác định, thì mọi ký tự trong bộ Unicode bắt buộc, khi được lưu trữ trong một đối tượng thuộc loại wchar\_t, có cùng giá trị với mã định danh ngắn của ký tự đó. Bộ Unicode bắt buộc bao gồm tất cả các ký tự được xác định bởi ISO/IEC 10646, cùng với tất cả các sửa đổi và sửa đổi kỹ thuật, kể từ năm và tháng được chỉ định. Nếu một số mã hóa khác được sử dụng, thì macro sẽ không được xác định và mã hóa thực tế được sử dụng là do triển khai xác định.
* \_\_STDC\_MB\_MIGHT\_NEQ\_WC\_\_ Hằng số nguyên 1, nhằm chỉ ra rằng, trong mã hóa cho wchar\_t, một thành viên của bộ ký tự cơ bản không cần có giá trị mã bằng với giá trị của nó khi được sử dụng làm ký tự đơn trong hằng ký tự số nguyên.
* \_\_STDC\_UTF\_16\_\_ Hằng số nguyên 1, nhằm biểu thị rằng các giá trị của loại char16\_tđược mã hóa UTF−16. Nếu một số mã hóa khác được sử dụng, thì macro sẽ không được xác định và mã hóa thực tế được sử dụng là do triển khai xác định.
* \_\_STDC\_UTF\_32\_\_ Hằng số nguyên 1, nhằm biểu thị rằng các giá trị của loại char32\_tđược mã hóa UTF−32. Nếu một số mã hóa khác được sử dụng, thì macro sẽ không được xác định và mã hóa thực tế được sử dụng là do triển khai xác định.
* ISO/IEC 9899:2011 §6.10.8.3 Macro tính năng có điều kiện
* \_\_STDC\_ANALYZABLE\_\_ Hằng số nguyên 1, nhằm biểu thị sự phù hợp với các thông số kỹ thuật trong phụ lục L (Khả năng phân tích).
* \_\_STDC\_IEC\_559\_\_ Hằng số nguyên 1, nhằm biểu thị sự phù hợp với các thông số kỹ thuật trong phụ lục F (số học dấu phẩy động IEC 60559).
* \_\_STDC\_IEC\_559\_COMPLEX\_\_Hằng số nguyên 1, nhằm biểu thị việc tuân thủ các thông số kỹ thuật trong phụ lục G (số học phức hợp tương thích với IEC 60559).
* \_\_STDC\_LIB\_EXT1\_\_Hằng số nguyên 201112L, nhằm biểu thị hỗ trợ cho các phần mở rộng được xác định trong phụ lục K (Giao diện kiểm tra giới hạn).
* \_\_STDC\_NO\_ATOMICS\_\_ Hằng số nguyên 1, nhằm chỉ ra rằng việc triển khai không hỗ trợ các loại nguyên tử (bao gồm cả bộ \_Atomic định loại) và <stdatomic.h>tiêu đề.
* \_\_STDC\_NO\_COMPLEX\_\_ Hằng số nguyên 1, nhằm chỉ ra rằng việc triển khai không hỗ trợ các loại phức tạp hoặc <complex.h> tiêu đề.
* \_\_STDC\_NO\_THREADS\_\_Hằng số nguyên 1, nhằm chỉ ra rằng việc triển khai không hỗ trợ <threads.h>tiêu đề.
* \_\_STDC\_NO\_VLA\_\_ Hằng số nguyên 1, nhằm biểu thị rằng việc triển khai không hỗ trợ các mảng có độ dài thay đổi hoặc các loại được sửa đổi thay đổi.

## MACRO VỚI ĐỐI SỐ BIẾN THIÊN

Phiên bản ≥ C99

Các macro với đối số biến thiên:

Hãy giả sử bạn muốn tạo một macro in ra thông tin để gỡ lỗi mã của bạn, hãy lấy ví dụ về macro sau đây:

|  |
| --- |
| #define debug\_print(*msg*) printf("%s:%d %s", \_\_FILE\_\_, \_\_LINE\_\_, msg) |

Một số ví dụ về cách sử dụng:

Hàm somefunc() trả về -1 nếu thất bại và 0 nếu thành công, và nó được gọi từ nhiều nơi khác nhau trong mã:

|  |
| --- |
| int retVal = somefunc();  if(retVal == -1)  {  debug\_printf("somefunc() has failed");  }  /\* some other code \*/  retVal = somefunc();  if(retVal == -1)  {  debug\_printf("somefunc() has failed");  } |

Điều gì sẽ xảy ra nếu triển khai của somefunc() thay đổi và giờ đây nó trả về các giá trị khác nhau phù hợp với các loại lỗi khác nhau có thể xảy ra? Bạn vẫn muốn sử dụng macro gỡ lỗi và in ra giá trị lỗi.

|  |
| --- |
| debug\_printf(retVal); /\* this would obviously fail \*/  debug\_printf("%d",retVal); /\* this would also fail \*/ |

Để giải quyết vấn đề này, \_\_VA\_ARGS\_\_macro đã được giới thiệu. Macro này cho phép nhiều tham số của X-macro:

Ví dụ:

|  |
| --- |
| #define debug\_print(*msg*, ...) printf(msg, \_\_VA\_ARGS\_\_) \  printf("\nError occurred in file:line (%s:%d)\n", \_\_FILE\_\_, \_\_LINE) |

Cách sử dụng:

|  |
| --- |
| int retVal = somefunc();  debug\_print("retVal of somefunc() is-> %d", retVal); |

Macro này cho phép bạn chuyển nhiều tham số và in chúng, nhưng hiện tại nó cấm bạn gửi bất kỳ tham số nào.

|  |
| --- |
| debug\_print("Hey"); |

Điều này sẽ gây ra một số lỗi cú pháp vì macro cần ít nhất một đối số nữa và bộ xử lý trước sẽ không bỏ qua việc thiếu dấu phẩy trong macro debug\_print(). Cũng debug\_print("Hey",);sẽ phát sinh lỗi cú pháp vì bạn không thể giữ đối số được chuyển sang macro trống.

Để giải quyết vấn đề này, ##\_\_VA\_ARGS\_\_macro đã được giới thiệu, macro này nói rằng nếu không có đối số biến nào tồn tại, thì dấu phẩy sẽ bị bộ xử lý trước xóa khỏi mã.

Ví dụ:

|  |
| --- |
| #define debug\_print(*msg*, ...) printf(msg, ##\_\_VA\_ARGS\_\_) \  printf("\nError occured in file:line (%s:%d)\n", \_\_FILE\_\_, \_\_LINE) |

Cách sử dụng:

|  |
| --- |
| debug\_print("Ret val of somefunc()?");  debug\_print("%d",somefunc()); |

## THAY THẾ MACRO

Dạng đơn giản nhất của việc thay thế Macro là định nghĩa một hằng số biểu tượng, như sau:

|  |
| --- |
| #define ARRSIZE 100  int array[ARRSIZE]; |

Đoạn mã trên định nghĩa một Macro hình thức hàm, nhân một biến với 10 và lưu giá trị mới:

|  |
| --- |
| #define TIMES10(*A*) ((A) \*= 10)  double b = 34;  int c = 23;  TIMES10(b); // tốt: ((b) \*= 10);  TIMES10(c); // tốt: ((c) \*= 10);  TIMES10(5); // không tốt: ((5) \*= 10); |

Việc thay thế được thực hiện trước bất kỳ sự diễn giải nào khác của văn bản chương trình. Trong lần gọi TIMES10 đầu tiên, tên A trong định nghĩa được thay thế bởi b và văn bản được mở rộng như vậy được đặt vào chỗ của lời gọi. Lưu ý rằng định nghĩa TIMES10 này không tương đương với

|  |
| --- |
| #define TIMES10(*A*) ((A) = (A) \* 10) |

bởi vì điều này có thể đánh giá thay thế của A hai lần, điều này có thể có tác động không mong muốn.

Dưới đây là định nghĩa một Macro hình thức hàm giá trị lớn nhất của các đối số. Nó có lợi thế làm việc với bất kỳ loại tương thích nào của các đối số và tạo mã trong-line mà không có gánh nặng của việc gọi hàm. Nó có nhược điểm là đánh giá một trong các đối số khác một lần nữa (bao gồm tác động phụ) và tạo ra nhiều mã hơn so với hàm nếu được gọi nhiều lần.

|  |
| --- |
| #define max(*a*, *b*) ((a) > (b) ? (a) : (b))  int maxVal = max(11, 43); /\* 43 \*/  int maxValExpr = max(11 + 36, 51 - 7); /\* 47 \*/  /\* Không nên làm, vì biểu thức được đánh giá hai lần \*/  int j = 0, i = 0;  int sideEffect = max(++i, ++j); /\* i == 4 \*/ |

Do đó, những Macro đánh giá đối số của chúng nhiều lần thường được tránh trong mã sản xuất. Từ phiên bản C11 trở đi, có tính năng \_Generic cho phép tránh những lời gọi nhiều lần như vậy.

Những dấu ngoặc đơn dư thừa trong các mở rộng Macro (phía bên phải của định nghĩa) đảm bảo rằng các đối số và biểu thức kết quả được ràng buộc đúng cách và phù hợp với ngữ cảnh trong đó Macro được gọi.

## CHỈ THỊ LỖI

Nếu trình biên dịch gặp một chỉ thị #error, quá trình biên dịch sẽ dừng lại và thông báo lỗi được bao gồm sẽ được in ra.

|  |
| --- |
| #define DEBUG  #ifdef DEBUG  #error "Debug Builds Not Supported"  #endif  int main(void) {  return 0;  } |

đầu ra có thể:

|  |
| --- |
| $ gcc error.c  error.c: error: #error "Debug Builds Not Supported" |

## KHAI TRIỂN FOREACH

Chúng ta cũng có thể sử dụng macro để làm cho mã dễ đọc và dễ viết hơn. Ví dụ: chúng ta có thể triển khai các macro để triển khai foreachcấu trúc trong C cho một số cấu trúc dữ liệu như danh sách liên kết đơn và đôi, hàng đợi, v.v.

Đây là một ví dụ nhỏ:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  struct *LinkedListNode*  {  int data;  struct *LinkedListNode* \*next;  };  #define FOREACH\_LIST(*node*, *list*) \  for (node=list; node; node=node->next)  /\* Usage \*/  int main(void)  {  struct *LinkedListNode* \*list, \*\*plist = &list, \*node;  int i;  for (i=0; i<10; i++)  {  \*plist = malloc(sizeof(struct *LinkedListNode*));  (\*plist)->data = i;  (\*plist)->next = NULL;  plist = &(\*plist)->next;  }  /\* printing the elements here \*/  FOREACH\_LIST(node, list)  {  printf("%d\n", node->data);  }  } |

Bạn có thể tạo một giao diện tiêu chuẩn cho các cấu trúc dữ liệu đó và viết cách triển khai chung FOREACH như:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  typedef struct *CollectionItem\_*  {  int data;  struct *CollectionItem\_* \*next;  } *CollectionItem*;  typedef struct *Collection\_*  {  /\* interface functions \*/  void\* (\*first)(void \*coll);  void\* (\*last) (void \*coll);  void\* (\*next) (void \*coll, *CollectionItem* \*currItem);  *CollectionItem* \*collectionHead;  /\* Other fields \*/  } *Collection*;  /\* must implement \*/  void \*first(void \**coll*)  {  return ((*Collection*\*)*coll*)->collectionHead;  }  /\* must implement \*/  void \*last(void \**coll*)  {  return NULL;  }  /\* must implement \*/  void \*next(void \**coll*, *CollectionItem* \**curr*)  {  return *curr*->next;  }  *CollectionItem* \*new\_CollectionItem(int *data*)  {  *CollectionItem* \*item = malloc(sizeof(*CollectionItem*));  item->data = *data*;  item->next = NULL;  return item;  }  void Add\_Collection(*Collection* \**coll*, int *data*)  {  *CollectionItem* \*\*item = &*coll*->collectionHead;  while(\*item)  item = &(\*item)->next;  (\*item) = new\_CollectionItem(*data*);  }  *Collection* \*new\_Collection()  {  *Collection* \*nc = malloc(sizeof(*Collection*));  nc->first = first;  nc->last = last;  nc->next = next;  return nc;  }  /\* generic implementation \*/  #define FOREACH(*node*, *collection*) \  for (node = (collection)->first(collection); \  node != (collection)->last(collection); \  node = (collection)->next(collection, node))  int main(void)  {  *Collection* \*coll = new\_Collection();  *CollectionItem* \*node;  int i;  for(i=0; i<10; i++)  {  Add\_Collection(coll, i);  }  /\* printing the elements here \*/  FOREACH(node, coll)  {  printf("%d\n", node->data);  }  } |

Để sử dụng triển khai chung này, chỉ cần triển khai các chức năng này cho cấu trúc dữ liệu của bạn.

|  |
| --- |
| 1. void\* (\*first)(void \*coll);  2. void\* (\*last) (void \*coll);  3. void\* (\*next) (void \*coll, CollectionItem \*currItem); |

# XỬ LÝ TÍN HIỆU

|  |  |
| --- | --- |
| Tham số | Chi tiết |
| sig | Tín hiệu để đặt bộ xử lý tín hiệu thành một trong SIGABRT, SIGFPE, SIGILL, SIGTERM, SIGINT hoặc SIGSEGV một số giá trị được xác định triển khai |
| func | Trình xử lý tín hiệu, là một trong hai cách sau: SIG\_DFL, đối với trình xử lý mặc định, SIG\_IGN để bỏ qua tín hiệu hoặc một con trỏ hàm có chữ ký void foo(int sig);. |

## XỬ LÝ TÍN HIỆU VỚI “SIGNAL()”

Các tín hiệu số có thể đồng bộ (như SIGSEGV– lỗi phân đoạn) khi chúng được kích hoạt bởi sự cố của chính chương trình hoặc không đồng bộ (như SIGINT- chú ý tương tác) khi chúng được bắt đầu từ bên ngoài chương trình, ví dụ: bằng cách nhấn phím như Ctrl-C.

Chức năng này signal() là một phần của tiêu chuẩn ISO C và có thể được sử dụng để gán chức năng xử lý một tín hiệu cụ thể

|  |
| --- |
| #include <stdio.h> /\* printf() \*/  #include <stdlib.h> /\* abort() \*/  #include <signal.h> /\* signal() \*/  void handler\_nonportable(int *sig*)  {  /\* undefined behavior, maybe fine on specific platform \*/  printf("Catched: %d\n", *sig*);    /\* abort is safe to call \*/  abort();  }  *sig\_atomic\_t* volatile finished = 0;  void handler(int *sig*)  {  switch (*sig*) {  /\* hardware interrupts should not return \*/  case SIGSEGV:  case SIGFPE:  case SIGILL:  C11  /\* quick\_exit is safe to call \*/  quick\_exit(EXIT\_FAILURE);  C11  /\* use \_Exit in pre-C11 \*/  \_Exit(EXIT\_FAILURE);  default:  /\* Reset the signal to the default handler,  so we will not be called again if things go  wrong on return. \*/  signal(*sig*, SIG\_DFL);  /\* let everybody know that we are finished \*/  finished = *sig*;  return;  }  }  int main(void)  {  /\* Catch the SIGSEGV signal, raised on segmentation faults (i.e NULL ptr access \*/  if (signal(SIGSEGV, &handler) == SIG\_ERR) {  perror("could not establish handler for SIGSEGV");  return EXIT\_FAILURE;  }  /\* Catch the SIGTERM signal, termination request \*/  if (signal(SIGTERM, &handler) == SIG\_ERR) {  perror("could not establish handler for SIGTERM");  return EXIT\_FAILURE;  }  /\* Ignore the SIGINT signal, by setting the handler to `SIG\_IGN`. \*/  signal(SIGINT, SIG\_IGN);  /\* Do something that takes some time here, and leaves  the time to terminate the program from the keyboard. \*/  /\* Then: \*/  if (finished) {  fprintf(stderr, "we have been terminated by signal %d\n", (int)finished);  return EXIT\_FAILURE;  }  /\* Try to force a segmentation fault, and raise a SIGSEGV \*/  {  char\* ptr = 0;  \*ptr = 0;  }  /\* This should never be executed \*/  return EXIT\_SUCCESS;  } |

Việc sử dụng signal()áp đặt những giới hạn quan trọng mà bạn được phép thực hiện bên trong bộ xử lý tín hiệu, hãy xem phần nhận xét để biết thêm thông tin.

POSIX khuyến nghị sử dụng sigaction()thay vì signal(), do hành vi chưa được chỉ định và các biến thể triển khai quan trọng của nó. POSIX cũng định nghĩa nhiều tín hiệu hơn tiêu chuẩn ISO C, bao gồm SIGUSR1và SIGUSR2, lập trình viên có thể sử dụng tự do cho bất kỳ mục đích nào.

# ĐỐI SỐ BIẾN

|  |  |
| --- | --- |
| Tham số | Chi tiết |
| va\_list ap | con trỏ đối số, vị trí hiện tại trong danh sách đối số biến đổi |
| last | tên của đối số hàm không biến đổi cuối cùng, để trình biên dịch tìm đúng vị trí để bắt đầu xử lý các đối số biến đổi; có thể không được khai báo dưới dạng một registerbiến, một hàm hoặc một kiểu mảng |
| type | loại đối số biến đổi được quảng cáoint để đọc (ví dụ đối với một short intđối số) |
| va\_list src | con trỏ đối số hiện tại để sao chép |
| va\_list dst | danh sách đối số mới được điền vào |

**Các đối số biến** được sử dụng bởi các hàm trong họ printf ( printf, fprintf, v.v.) và các hàm khác để cho phép một hàm được gọi với số lượng đối số khác nhau mỗi lần, do đó có tên varargs (biến số đối số)

Để triển khai các hàm sử dụng tính năng biến đối số, sử dụng #include <stdarg.h>.

Để gọi các hàm có chấp nhận số lượng đối số biến đổi, đảm bảo rằng có một nguyên mẫu đầy đủ với dấu ba chấm ở cuối: ví dụ void err\_exit(const char \*format, ...);

## SỬ DỤNG ĐỐI SỐ ĐẾM RÕ RÀNG ĐỂ XÁC ĐỊNH ĐỘ DÀI CỦA VA\_LIST

Với bất kỳ hàm biến đổi nào, hàm phải biết cách giải thích danh sách đối số biến đổi. Với các hàm printf() hoặc scanf(), chuỗi định dạng nói cho hàm biết điều gì cần mong đợi.

Kỹ thuật đơn giản nhất là chuyển một đếm rõ ràng của các đối số khác (mà thường là cùng loại).

Điều này được thể hiện trong hàm biến đối số trong mã dưới đây, tính tổng của một loạt các số nguyên, nơi có thể có bất kỳ số nguyên nào nhưng số lượng đó được chỉ định là một đối số trước danh sách đối số biến đổi.

|  |
| --- |
| #include <stdio.h>  #include <stdarg.h>  /\* đối số đầu tiên là số lượng đối số int tiếp theo cần tổng. \*/  int sum(int *n*, ...) {  int sum = 0;  *va\_list* it; /\* lưu thông tin về danh sách đối số biến đổi. \*/  va\_start(it, *n*); /\* bắt đầu xử lý đối số biến đổi \*/  while (*n*--)  sum += va\_arg(it, int); /\* lấy và cộng các đối số biến đổi tiếp theo \*/  va\_end(it); /\* kết thúc xử lý đối số biến đổi \*/  return sum;  }  int main(void)  {  printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /\* in ra 15 \*/  printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /\* in ra 666 \*/  return 0;  } |

## SỬ DỤNG GIÁ TRỊ KẾT THÚC ĐỂ XÁC ĐỊNH PHẦN CUỐI CỦA VA\_LIST

Với bất kỳ hàm biến đổi nào, hàm phải biết cách giải thích danh sách đối số biến đổi. Phương pháp "truyền thống" (được minh họa bởi printf) là xác định số lượng đối số từ đầu. Tuy nhiên, điều này không phải lúc nào cũng là một ý tưởng tốt:

|  |
| --- |
| /\* Đối số đầu tiên xác định số lượng tham số; số còn lại cũng là int \*/  extern int sum(int *n*, ...);  /\* Nhưng điều này không rõ ràng từ code. \*/  sum(5, 2, 1, 4, 3, 6)  /\* Điều gì sẽ xảy ra nếu một đối số được loại bỏ sau này? \*/  sum(5, 2, 1, 3, 6) /\* Thảm họa \*/ |

Đôi khi, việc thêm một trình kết thúc rõ ràng, được minh họa bằng execlp() hàm POSIX, sẽ hiệu quả hơn. Đây là một hàm khác để tính tổng của một loạt các số thực

|  |
| --- |
| #include <stdarg.h>  #include <stdio.h>  #include <math.h>  /\* Cộng các đối số cho đến giá trị kết thúc NAN \*/  double sum (double *x*, ...) {  double sum = 0;  *va\_list* va;  va\_start(va, *x*);  for (; !isnan(*x*); *x* = va\_arg(va, double)){  sum += *x*;  }  va\_end(va);  return sum;  }  int main (void) {  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));  } |

Một số giá trị kết thúc tốt:

* Kiểu số nguyên (giả định là tất cả đều là số dương hoặc không âm) - 0 hoặc -1
* Các kiểu số dấu chấm động - NAN
* Các kiểu con trỏ - NULL
* Các kiểu enumerator - một số giá trị đặc biệt

## TRIỂN KHAI CÁC HÀM VỚI GIAO DIỆN GIỐNG NHƯ ‘ PRINTF() ’

Một trong những cách thường thấy sử dụng danh sách đối số biến đổi là triển khai các hàm mỏng manh bọc xung quanh gia đình hàm printf(). Một ví dụ như vậy là một tập hợp các hàm báo cáo lỗi.

**errmsg.h**

|  |
| --- |
| #ifndef ERRMSG\_H\_INCLUDED  #define ERRMSG\_H\_INCLUDED  #include <stdarg.h>  #include <stdnoreturn.h> // C11  void verrmsg(int *errnum*, const char \**fmt*, *va\_list* *ap*);  noreturn void errmsg(int *exitcode*, int *errnum*, const char \**fmt*, ...);  void warnmsg(int *errnum*, const char \**fmt*, ...);  #endif |

Đây là một ví dụ cơ bản; những gói như vậy có thể phức tạp hơn. Thông thường, người lập trình sẽ sử dụng hàm errmsg() hoặc warnmsg(), chính chúng sử dụng verrmsg() bên trong. Nếu ai đó cần làm nhiều hơn, thì hàm verrmsg() được hiển thị sẽ hữu ích. Bạn có thể tránh hiển thị nó cho đến khi bạn cần (YAGNI - bạn không cần nó), nhưng cần sẽ xuất hiện sớm hoặc muộn (bạn sẽ cần nó - YAGNI).

**errmsg.c**

Đoạn mã này chỉ cần chuyển các đối số biến đổi cho hàm vfprintf() để in ra tiêu chuẩn lỗi. Nó cũng thông báo về thông báo lỗi hệ thống tương ứng với số lỗi hệ thống (errno) được chuyển đến các hàm.

|  |
| --- |
| #include "errmsg.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  void verrmsg(int *errnum*, const char \**fmt*, *va\_list* *ap*)  {  if (*fmt*)  vfprintf(stderr, *fmt*, *ap*);  if (*errnum* != 0)  fprintf(stderr, ": %s", strerror(*errnum*));  putc('\n', stderr);  }  void errmsg(int *exitcode*, int *errnum*, const char \**fmt*, ...)  {  *va\_list* ap;  va\_start(ap, *fmt*);  verrmsg(*errnum*, *fmt*, ap);  va\_end(ap);  exit(*exitcode*);  }  void warnmsg(int *errnum*, const char \**fmt*, ...)  {  *va\_list* ap;  va\_start(ap, *fmt*);  verrmsg(*errnum*, *fmt*, ap);  va\_end(ap);  } |

**Sử dụng errmsg.h**

Bây giờ bạn có thể sử dụng các hàm như sau:

|  |
| --- |
| #include "errmsg.h"  #include <errno.h>  #include <fcntl.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  int main(int *argc*, char \*\**argv*)  {  char buffer[BUFSIZ];  int fd;  if (*argc* != 2)  {  fprintf(stderr, "Usage: %s filename\n", *argv*[0]);  exit(EXIT\_FAILURE);  }  const char \*filename = *argv*[1];  if ((fd = open(filename, O\_RDONLY)) == -1)  errmsg(EXIT\_FAILURE, errno, "cannot open %s", filename);  if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))  errmsg(EXIT\_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer), filename);  if (close(fd) == -1)  warnmsg(errno, "cannot close %s", filename);  /\* tiếp tục chương trình \*/  return 0;  } |

Nếu lệnh gọi open() hoặc read() hệ thống không thành công, lỗi được ghi thành lỗi tiêu chuẩn và chương trình thoát với mã thoát 1. Nếu lệnh close() gọi hệ thống không thành công, lỗi chỉ được in dưới dạng thông báo cảnh báo và chương trình sẽ tiếp tục.

**Kiểm tra việc sử dụng đúng các định dạng printf()**

Nếu bạn đang sử dụng GCC (Trình biên dịch GNU C, là một phần của Bộ sưu tập Trình biên dịch GNU) hoặc sử dụng Clang, thì bạn có thể yêu cầu trình biên dịch kiểm tra xem các đối số mà bạn chuyển đến các hàm thông báo lỗi có khớp với những gì mong đợi hay không printf(). Vì không phải tất cả các trình biên dịch đều hỗ trợ phần mở rộng, nên nó cần được biên dịch theo điều kiện, điều này hơi khó một chút. Tuy nhiên, sự bảo vệ mà nó mang lại rất đáng để nỗ lực.

Trước tiên, chúng ta cần biết cách phát hiện trình biên dịch là GCC hoặc Clang mô phỏng GCC. Câu trả lời là GCC định nghĩa \_\_GNUC\_\_để chỉ ra điều đó.

Xem các thuộc tính chức năng phổ biến để biết thông tin về các thuộc tính — cụ thể là format thuộc tính.

Viết lại **errmsg.h**

|  |
| --- |
| #ifndef ERRMSG\_H\_INCLUDED  #define ERRMSG\_H\_INCLUDED  #include <stdarg.h>  #include <stdnoreturn.h> // C11  #if !defined(PRINTFLIKE)  #if defined(\_\_GNUC\_\_)  #define PRINTFLIKE(*n*,*m*) \_\_attribute\_\_((format(printf,n,m)))  #else  #define PRINTFLIKE(*n*,*m*) /\* If only \*/  #endif /\* \_\_GNUC\_\_ \*/  #endif /\* PRINTFLIKE \*/  void verrmsg(int *errnum*, const char \**fmt*, va\_list *ap*);  void noreturn errmsg(int *exitcode*, int *errnum*, const char \**fmt*, ...)  PRINTFLIKE(3, 4);  void warnmsg(int *errnum*, const char \**fmt*, ...)  PRINTFLIKE(2, 3);  #endif |

Bây giờ, nếu bạn mắc lỗi như:

|  |
| --- |
| rrmsg(EXIT\_FAILURE, errno, "Failed to open file '%d' for reading", filename); |

(nơi %d nên là %s), thì trình biên dịch sẽ phàn nàn:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \  > -Wold-style-definition -c erruse.c  erruse.c: In function ‘main’:  erruse.c:20:64: error: format ‘%d’ expects argument of type ‘int’, but argument 4 has type ‘const char \*’ [-Werror=format=]  errmsg(EXIT\_FAILURE, errno, "Failed to open file '%d' for reading", filename);  ~^  %s  cc1: all warnings being treated as errors  $ |

## SỬ DỤNG CHUỖI ĐỊNH DẠNG

Việc sử dụng một chuỗi định dạng cung cấp thông tin về số lượng và loại dự kiến ​​của các đối số biến thiên tiếp theo theo cách để tránh nhu cầu về đối số đếm rõ ràng hoặc giá trị của dấu kết thúc.

Ví dụ dưới đây thể hiện một hàm bọc chức năng printf() tiêu chuẩn, chỉ cho phép sử dụng các đối số biến đổi có kiểu char, int và double (trong định dạng điểm động thập phân thập phân). Ở đây, tương tự như với printf(), đối số đầu tiên của hàm bọc là chuỗi định dạng. Khi chuỗi định dạng được phân tích cú pháp, hàm có thể xác định xem có một đối số biến đổi khác được mong đợi và loại của nó là gì.

|  |
| --- |
| #include <stdio.h>  #include <stdarg.h>  int simple\_printf(const char \**format*, ...)  {  *va\_list* ap; /\* lưu thông tin về danh sách đối số biến đổi \*/  int printed = 0; /\* đếm số ký tự được in \*/  va\_start(ap, *format*); /\* bắt đầu xử lý đối số biến đổi \*/  while (\**format* != '\0') /\* đọc chuỗi định dạng cho đến ký tự kết thúc chuỗi \*/  {  int f = 0;  if (\**format* == '%')  {  ++*format*;  switch(\**format*)  {  case 'c' :  f = printf("%d", va\_arg(ap, int)); /\* in đối số biến đổi tiếp theo, lưu ý kiểu  thăng cấp từ char thành int \*/  break;  case 'd' :  f = printf("%d", va\_arg(ap, int)); /\* in đối số biến đổi tiếp theo \*/  break;  case 'f' :  f = printf("%f", va\_arg(ap, double)); /\* in đối số biến đổi tiếp theo \*/  break;  default :  f = -1; /\* thông báo định dạng không hợp lệ \*/  break;  }  }  else  {  f = printf("%c", \**format*); /\* in các ký tự khác \*/  }  if (f < 0) /\* kiểm tra lỗi \*/  {  printed = f;  break;  }  else  {  printed += f;  }  ++*format*; /\* tiếp tục đến ký tự tiếp theo trong chuỗi \*/  }  va\_end(ap); /\* kết thúc xử lý đối số biến đổi \*/  return printed;  }  int main (int *argc*, char \**argv*[])  {  int x = 40;  int y = 0;  y = simple\_printf("There are %d characters in this sentence", x);  simple\_printf("\n%d were printed\n", y);  } |

# KHẲNG ĐỊNH

|  |  |
| --- | --- |
| **Tham số** | **Chi tiết** |
| expression | Biểu thức có kiểu scalar. |
| message | chuỗi ký tự được bao gồm trong thông báo chẩn đoán. |

Một khẳng định là một tiền đề rằng điều kiện đã được trình bày phải là đúng tại thời điểm khẳng định được gặp bởi phần mềm. Thông thường, khẳng định đơn giản là các kiểm tra được thực hiện tại thời điểm thực thi. Tuy nhiên, khẳng định tĩnh được kiểm tra tại thời gian biên dịch.

## KHẲNG ĐỊNH ĐƠN GIẢN

Khẳng định là một câu lệnh được sử dụng để khẳng định rằng một sự thật phải đúng khi dòng mã đó đến. Khẳng định rất hữu ích để đảm bảo các điều kiện mong đợi được đáp ứng. Khi điều kiện được chuyển vào một khẳng định là đúng, không có hành động nào được thực hiện. Hành vi trên điều kiện sai phụ thuộc vào các cờ trình biên dịch. Khi khẳng định được kích hoạt, đầu vào sai sẽ dẫn đến dừng chương trình ngay lập tức. Khi chúng được vô hiệu hóa, không có hành động nào được thực hiện. Thông thường, thực hành phổ biến là bật khẳng định trong các phiên bản nội bộ và gỡ lỗi, và tắt chúng trong các phiên bản phát hành, mặc dù các khẳng định thường được bật trong các phiên bản phát hành. (Việc dừng chương trình tốt hơn hay xấu hơn lỗi phụ thuộc vào chương trình.) Khẳng định nên chỉ được sử dụng để phát hiện lỗi lập trình nội bộ, điều này thường đồng nghĩa với việc chuyển các tham số không hợp lệ.

|  |
| --- |
| #include <stdio.h>  /\* Uncomment to disable `assert()` \*/  /\* #define NDEBUG \*/  #include <assert.h>  int main(void)  {  int x = -1;  assert(x >= 0);  printf("x = %d\n", x);  return 0;  } |

Kết quả có thể có khi NDEBUG không được định nghĩa:

|  |
| --- |
| a.out: main.c:9: main: Assertion `x >= 0' failed. |

Kết quả có thể có khi NDEBUG được định nghĩa:

|  |
| --- |
| x = -1 |

Việc thực hành tốt là định nghĩa NDEBUG một cách toàn cầu, để bạn có thể dễ dàng biên dịch mã của mình với tất cả các khẳng định bật hoặc tắt. Một cách dễ dàng để làm điều này là định nghĩa NDEBUG như một tùy chọn cho trình biên dịch, hoặc định nghĩa nó trong một tiêu đề cấu hình chia sẻ (ví dụ: config.h).

## KHẲNG ĐỊNH TĨNH

Phiên bản ≥ C11

Khẳng định tĩnh được sử dụng để kiểm tra xem điều kiện có đúng khi mã được biên dịch hay không. Nếu không đúng, trình biên dịch sẽ phải phát ra thông báo lỗi và dừng quá trình biên dịch.

Khẳng định tĩnh là một kiểm tra được thực hiện tại thời gian biên dịch, không phải thời gian chạy. Điều kiện phải là một biểu thức hằng số, và nếu sai sẽ dẫn đến lỗi trình biên dịch. Đối số đầu tiên, điều kiện được kiểm tra, phải là một biểu thức hằng số, và tham số thứ hai là một chuỗi ký tự cố định.

Khác với **assert**, **\_Static\_assert** là một từ khóa. Một tiện ích macro **static\_assert** được định nghĩa trong **<assert.h>.**

|  |
| --- |
| #include <assert.h>  enum {N = 5};  \_Static\_assert(N == 5, "N does not equal 5");  static\_assert(N > 10, "N is not greater than 10"); /\* lỗi biên dịch \*/ |

Phiên bản = C99

Trước C11, không có hỗ trợ trực tiếp cho khẳng định tĩnh. Tuy nhiên, trong C99, khẳng định tĩnh có thể được mô phỏng bằng các macro sẽ kích hoạt lỗi biên dịch nếu điều kiện thời gian biên dịch là sai. Khác với \_Static\_assert, tham số thứ hai cần là một tên mã thích hợp để có thể tạo ra một tên biến từ nó. Nếu khẳng định thất bại, tên biến sẽ được thấy trong thông báo lỗi của trình biên dịch, vì biến đó đã được sử dụng trong một khai báo mảng cú pháp không chính xác.

|  |
| --- |
| #define STATIC\_MSG(*msg*, *l*) STATIC\_MSG2(msg, l)  #define STATIC\_MSG2(*msg*,*l*) on\_line\_##l##\_\_##msg  #define STATIC\_ASSERT(*x*, *msg*) extern char STATIC\_MSG(msg, \_\_LINE\_\_) [(x)?1:-1]  enum { N = 5 };  STATIC\_ASSERT(N == 5, N\_must\_equal\_5);  STATIC\_ASSERT(N > 5, N\_must\_be\_greater\_than\_5); /\* lỗi biên dịch \*/ |

Trước C99, bạn không thể khai báo biến tại các vị trí tùy ý trong một khối, vì vậy bạn phải cực kỳ cẩn trọng khi sử dụng macro này, đảm bảo rằng nó chỉ xuất hiện ở nơi mà một khai báo biến sẽ hợp lệ

## KHẲNG ĐỊNH TIN NHẮN LỖI

Tồn tại một thủ thuật có thể hiển thị thông báo lỗi cùng với xác nhận. Thông thường, bạn sẽ viết mã như thế này

|  |
| --- |
| void f(void \**p*)  {  assert(*p* != NULL);  /\* more code \*/  } |

Nếu xác nhận không thành công, một thông báo lỗi sẽ giống như

Assertion failed: p != NULL, file main.c, line 5

Tuy nhiên, bạn cũng có thể sử dụng logic AND (&&) để đưa ra thông báo lỗi

|  |
| --- |
| void f(void \**p*)  {  assert(*p* != NULL && "function f: p cannot be NULL");  /\* more code \*/  } |

Bây giờ, nếu xác nhận không thành công, một thông báo lỗi sẽ đọc như thế này

Assertion failed: p != NULL && "function f: p cannot be NULL", file main.c, line 5

Lý do tại sao điều này hoạt động là một chuỗi ký tự luôn đánh giá khác không (đúng). Việc thêm && 1 vào một biểu thức Boolean không có tác dụng. Do đó, việc thêm vào && "**error message**"cũng không có tác dụng gì, ngoại trừ việc trình biên dịch sẽ hiển thị toàn bộ biểu thức bị lỗi.

## KHẲNG ĐỊNH MÃ KHÔNG THỂ TRUY CẬP ĐƯỢC

Trong quá trình phát triển, khi một số lối đi mã phải được ngăn cản khỏi việc kiểm soát luồng, bạn có thể sử dụng assert(0) để chỉ ra rằng điều kiện như vậy là sai:

|  |
| --- |
| switch (color) {  case COLOR\_RED:  case COLOR\_GREEN:  case COLOR\_BLUE:  break;  default:  assert(0);  } |

Khi đối số của macro assert() được đánh giá sai, macro sẽ viết thông tin chẩn đoán vào luồng lỗi chuẩn và sau đó dừng chương trình. Thông tin này bao gồm tệp và số dòng của câu lệnh assert() và có thể rất hữu ích trong gỡ lỗi. Khẳng định có thể bị vô hiệu hóa bằng cách định nghĩa macro NDEBUG.

Một cách khác để kết thúc chương trình khi xảy ra lỗi là sử dụng các hàm thư viện tiêu chuẩn exit, quick\_exit hoặc abort. exit và quick\_exit lấy một đối số có thể được truyền lại cho môi trường của bạn. abort() (và do đó assert) có thể là một kết thúc rất nghiêm trọng của chương trình của bạn, và một số công việc dọn dẹp mà sẽ được thực hiện ở cuối thực thi, có thể không được thực hiện.

Lợi thế chính của assert() là nó tự động in thông tin gỡ lỗi. Gọi abort() có lợi thế là không thể vô hiệu hóa như một assert, nhưng nó có thể không gây ra bất kỳ thông tin gỡ lỗi nào được hiển thị. Trong một số tình huống, sử dụng cả hai cấu trúc cùng nhau có thể có lợi:

|  |
| --- |
| if (color == COLOR\_RED || color == COLOR\_GREEN) {  ...  } else if (color == COLOR\_BLUE) {  ...  } else {  assert(0), abort();  } |

Khi asserts được kích hoạt, cuộc gọi assert() sẽ in thông tin gỡ lỗi và kết thúc chương trình. Thực thi không bao giờ đạt đến cuộc gọi abort(). Khi asserts được vô hiệu hóa, cuộc gọi assert() không làm gì cả và abort() được gọi. Điều này đảm bảo rằng chương trình luôn kết thúc cho điều kiện lỗi này; kích hoạt và vô hiệu hóa asserts chỉ ảnh hưởng đến việc có hay không có đầu ra gỡ lỗi được in.

Bạn không bao giờ nên để một assert như vậy trong mã sản xuất, vì thông tin gỡ lỗi không hữu ích cho người dùng cuối và vì abort thường là một cách kết thúc quá mức nghiêm trọng và ngăn cản các trình xử lý dọn dẹp được cài đặt để chạy cho exit hoặc quick\_exit.

## TIỀN ĐIỀU KIỆN VÀ HẬU ĐIỀU KIỆN

Một trong những trường hợp sử dụng cho khẳng định là kiểm tra điều kiện tiên quyết và điều kiện hậu quả. Điều này rất hữu ích để duy trì các điều kiện bất biến và thiết kế theo hợp đồng. Ví dụ, một độ dài luôn luôn là số không hoặc dương, do đó hàm này phải trả về một số không hoặc dương.

|  |
| --- |
| #include <stdio.h>  /\* Uncomment to disable `assert()` \*/  /\* #define NDEBUG \*/  #include <assert.h>  int length2 (int \**a*, int *count*)  {  int i, result = 0;  /\* Điều kiện tiên quyết: \*/  /\* NULL là một vector không hợp lệ \*/  assert (*a* != NULL);  /\* Số chiều không thể là số âm. \*/  assert (*count* >= 0);  /\* Tính toán \*/  for (i = 0; i < *count*; ++i)  {  result = result + (*a*[i] \* *a*[i]);  }  /\* Điều kiện hậu quả: \*/  /\* Độ dài kết quả không thể là số âm. \*/  assert (result >= 0);  return result;  }  #define COUNT 3  int main (void)  {  int a[COUNT] = {1, 2, 3};  int \*b = NULL;  int r;  r = length2 (a, COUNT);  printf ("r = %i\n", r);  r = length2 (b, COUNT);  printf ("r = %i\n", r);  return 0;  } |

# LỰA CHỌN TỔNG QUÁT

|  |  |
| --- | --- |
| Tham số | Chi tiết |
| generic-assoc-list | generic-association OR generic-assoc-list , generic-association |
| generic-association | type-name : assignment-expression OR default : assignment-expression |

## KIỂM TRA XEM BIẾN CÓ PHẢI LÀ MỘT KIỂU CỤ THỂ ĐƯỢC CHÚ THÍCH

Mã chương trình dưới đây dùng \_Generic macro để kiểm tra xem biến có phải là kiểu con trỏ int hằng hay kiểu con trỏ int không phải hằng, và đưa ra thông báo tương ứng:

|  |
| --- |
| #include <stdio.h>  #define is\_const\_int(*x*) \_Generic((&x), \  const int \*: "a const int", \  int \*: "a non-const int", \  default: "of other type")\  int main(void)  {  const int i = 1;  int j = 1;  double k = 1.0;  printf("i is %s\n", is\_const\_int(i));  printf("j is %s\n", is\_const\_int(j));  printf("k is %s\n", is\_const\_int(k));  } |

Kết quả

|  |
| --- |
| i is a const int  j is a non-const int  k is of other type |

Tuy nhiên, nếu macro generic type được triển khai như sau:

|  |
| --- |
| #define is\_const\_int(*x*) \_Generic((x), \  const int: "a const int", \  int: "a non-const int", \  default: "of other type") |

Kết quả sẽ là

|  |
| --- |
| i is a non-const int  j is a non-const int  k is of other type |

Điều này xảy ra vì tất cả các kiểu bổ sung đều bị loại bỏ khi đánh giá biểu thức điều khiển trong một \_Generic primary expression.

## LỰA CHỌN THÔNG QUA KIỂU DỮ LIỆU CHUNG DỰA TRÊN NHIỀU ĐỐI SỐ

Nếu muốn thực hiện lựa chọn dựa trên nhiều đối số cho biểu thức kiểu dữ liệu chung, và tất cả các kiểu dữ liệu đều là kiểu số học, một cách dễ dàng để tránh việc nhúng các biểu thức \_Generic lồng nhau là sử dụng phép cộng của các tham số trong biểu thức kiểm soát:

|  |
| --- |
| int max\_int(int, int);  unsigned max\_unsigned(unsigned, unsigned);  double max\_double(double, double);  #define MAX(*X*, *Y*) \_Generic((X)+(Y), \  int: max\_int, \  unsigned: max\_unsigned, \  default: max\_double) \  ((X), (Y)) |

Ở đây, biểu thức kiểm soát (X)+(Y) chỉ được kiểm tra theo kiểu dữ liệu và không được đánh giá. Các chuyển đổi thông thường cho các toán hạng số học được thực hiện để xác định kiểu dữ liệu được chọn.

Để giải quyết tình huống phức tạp hơn, lựa chọn có thể được thực hiện dựa trên nhiều hơn một đối số của toán tử, bằng cách lồng chúng lại với nhau.

Ví dụ này lựa chọn giữa bốn hàm được triển khai bên ngoài, lấy các kết hợp của hai đối số int và/ hoặc string, và trả về tổng của chúng.

|  |
| --- |
| int AddIntInt(int *a*, int *b*);  int AddIntStr(int *a*, const char\* *b*);  int AddStrInt(const char\* *a*, int *b* );  int AddStrStr(const char\* *a*, const char\* *b*);  #define AddStr(*y*) \  \_Generic((y), int: AddStrInt, \  char\*: AddStrStr, \  const char\*: AddStrStr )  #define AddInt(*y*) \  \_Generic((y), int: AddIntInt, \  char\*: AddIntStr, \  const char\*: AddIntStr )  #define Add(*x*, *y*) \  \_Generic((x) , int: AddInt(y) , \  char\*: AddStr(y) , \  const char\*: AddStr(y)) \  ((x), (y))  int main( void )  {  int result = 0;  result = Add( 100 , 999 );  result = Add( 100 , "999" );  result = Add( "100" , 999 );  result = Add( "100" , "999" );  const int a = -123;  char b[] = "4321";  result = Add( a , b );  int c = 1;  const char d[] = "0";  result = Add( d , ++c );  } |

Mặc dù có vẻ như đối số yđược đánh giá nhiều lần, nhưng đó không phải là 1 . Cả hai đối số chỉ được đánh giá một lần, ở cuối macro Add: ( x , y ), giống như trong một lệnh gọi hàm thông thường.

1 (Trích dẫn từ: ISO:IEC 9899:201X 6.5.1.1 Lựa chọn chung 3)

Biểu thức kiểm soát của lựa chọn chung không được đánh giá.

## LOẠI MACRO IN CHUNG

|  |
| --- |
| #include <stdio.h>  void print\_int(int *x*) { printf("int: %d\n", *x*); }  void print\_dbl(double *x*) { printf("double: %g\n", *x*); }  void print\_default() { puts("unknown argument"); }  #define print(*X*) \_Generic((X), \  int: print\_int, \  double: print\_dbl, \  default: print\_default)(X)  int main(void) {  print(42);  print(3.14);  print("hello, world");  } |

Kết quả

|  |
| --- |
| int: 42  double: 3.14  unknown argument |

Lưu ý rằng nếu kiểu không phải là int hoặc double, một cảnh báo sẽ được tạo ra. Để loại bỏ cảnh báo, bạn có thể thêm kiểu đó vào macro print(X).

# X- MACROS

X-macro là một kỹ thuật dựa trên bộ tiền xử lý để giảm thiểu mã lặp lại và duy trì sự tương ứng giữa dữ liệu/mã. Nhiều phần mở rộng macro riêng biệt dựa trên một tập hợp dữ liệu chung được hỗ trợ bằng cách đại diện cho toàn bộ nhóm phần mở rộng thông qua một macro chính duy nhất, với văn bản thay thế của macro đó bao gồm một chuỗi các phần mở rộng của macro bên trong, mỗi phần mở rộng cho mỗi mốc dữ liệu. Macro trong thực tế thường được đặt tên là X(), vì vậy được gọi là X macros.

## SỬ DỤNG ĐƠN GIẢN CỦA X- MACROS CHO CÁC CÂU LỆNH PRINTF()

|  |
| --- |
| /\* Define một danh sách các mã thông báo trình biên dịch mà X sẽ được gọi tên \*/  #define X\_123 X(1) X(2) X(3)  /\* Define X mà sẽ được sử dụng \*/  #define X(*val*) printf("X(%d) made this print\n", val);  X\_123  #undef X  /\* Thói quen tốt là undef X để dễ dàng sử dụng lại sau này \*/ |

Ví dụ này sẽ dẫn đến trình biên dịch sinh ra mã sau đây:

|  |
| --- |
| printf("X(%d) thực hiện in này\n", 1);  printf("X(%d) thực hiện in này\n", 2);  printf("X(%d) thực hiện in này\n", 3); |

## MỞ RỘNG : SỬ DỤNG X- MACRO NHƯ LÀ ĐỐI SỐ

Cách tiếp cận X-macro có thể được khái quát hóa một chút bằng cách đặt tên của macro "X" thành đối số của macro chính. Điều này có ưu điểm là giúp tránh xung đột tên macro và cho phép sử dụng macro có mục đích chung làm macro "X".

Như mọi khi với macro X, macro chính đại diện cho danh sách các mục có ý nghĩa cụ thể đối với macro đó. Trong biến thể này, một macro như vậy có thể được định nghĩa như sau:

|  |
| --- |
| /\* declare list of items \*/  #define ITEM\_LIST(*X*) \  X(item1) \  X(item2) \  X(item3) \  /\* end of list \*/ |

Sau đó, người ta có thể tạo mã để in tên vật phẩm như vậy:

|  |
| --- |
| /\* define macro to apply \*/  #define PRINTSTRING(*value*) printf( #value "\n");  /\* apply macro to the list of items \*/  ITEM\_LIST(PRINTSTRING) |

Điều đó mở rộng sang mã này:

|  |
| --- |
| printf( "item1" "\n"); printf( "item2" "\n"); printf( "item3" "\n"); |

Trái ngược với macro X tiêu chuẩn, trong đó tên "X" là một đặc điểm tích hợp sẵn của macro chính, với kiểu này, có thể không cần thiết hoặc thậm chí không mong muốn sau đó xác định macro được sử dụng làm đối số ( trong ví dụ này) **PRINTSTRING.**

## GIÁ TRỊ ENUM VÀ MÃ ĐỊNH DANH

|  |
| --- |
| /\* declare items of the enum \*/  #define FOREACH \  X(item1) \  X(item2) \  X(item3) \  /\* end of list \*/  /\* define the enum values \*/  #define X(*id*) MyEnum\_ ## id,  enum *MyEnum* { FOREACH };  #undef X  /\* convert an enum value to its identifier \*/  const char \* enum2string(int *enumValue*)  {  const char\* stringValue = NULL;  #define X(*id*) if (*enumValue* == MyEnum\_ ## id) stringValue = #id;  FOREACH  #undef X  return stringValue;  } |

Tiếp theo, bạn có thể sử dụng giá trị liệt kê trong mã của mình và dễ dàng in mã định danh của nó bằng cách sử dụng:

|  |
| --- |
| printf("%s\n", enum2string(*MyEnum\_item2*)); |

## TẠO MÃ

X-Macro có thể được sử dụng để tạo mã, bằng cách viết mã lặp đi lặp lại: lặp qua một danh sách để thực hiện một số công việc hoặc để khai báo một tập hợp các hằng số, đối tượng hoặc hàm.

**Ở đây, chúng ta sử dụng X-Macro để khai báo một enum chứa 4 lệnh và một bảng ánh xạ tên của chúng thành chuỗi.**

Sau đó, chúng ta có thể in ra các giá trị chuỗi của enum.

|  |
| --- |
| /\* All our commands \*/  #define COMMANDS(*OP*) OP(Open) OP(Close) OP(Save) OP(Quit)  /\* generate the enum Commands: {cmdOpen, cmdClose, cmdSave, cmdQuit, }; \*/  #define ENUM\_NAME(*name*) cmd##name,  enum *Commands* {  COMMANDS(ENUM\_NAME)  };  #undef ENUM\_NAME  /\* generate the string table \*/  #define COMMAND\_OP(*name*) #name,  const char\* const commandNames[] = {  COMMANDS(COMMAND\_OP)  };  #undef COMMAND\_OP  /\* the following prints "Quit\n": \*/  printf("%s\n", commandNames[cmdQuit]()); |

**Tương tự, chúng ta có thể tạo một bảng nhảy (jump table) để gọi các hàm dựa trên giá trị của enum.**

Điều này yêu cầu tất cả các hàm có cùng chữ ký. Nếu chúng không có đối số và trả về một giá trị int, chúng ta sẽ đặt nó vào một tệp tiêu đề với định nghĩa enum.

|  |
| --- |
| /\* declare all functions as extern \*/  #define EXTERN\_FUNC(*name*) extern int doCmd##name(void);  COMMANDS(EXTERN\_FUNC)  #undef EXTERN\_FUNC  /\* declare the function pointer type and the jump table \*/  typedef int (\**CommandFunc*)(void);  extern *CommandFunc* commandJumpTable[]; |

Tất cả những điều sau đây có thể nằm trong các đơn vị biên dịch khác nhau giả sử phần trên được bao gồm dưới dạng tiêu đề:

|  |
| --- |
| /\* generate the jump table \*/  #define FUNC\_NAME(*name*) doCmd##name,  *CommandFunc* commandJumpTable[] = {  COMMANDS(FUNC\_NAME)  };  #undef FUNC\_NAME  /\* call the save command like this: \*/  int result = commandJumpTable[cmdSave]();  /\* somewhere else, we need the implementations of the commands \*/  int doCmdOpen(void) {/\* code performing open command \*/}  int doCmdClose(void) {/\* code performing close command \*/}  int doCmdSave(void) {/\* code performing save command \*/}  int doCmdQuit(void) {/\* code performing quit command \*/} |

Một ví dụ về kỹ thuật này đang được sử dụng trong mã thực là GPU command dispatching in Chromium.

# ALIASING VÀ EFFECTIVE TYPE

## EFFECTIVE TYPE

Effective type của một đối tượng dữ liệu là loại thông tin cuối cùng được liên kết với nó, nếu có.

|  |
| --- |
| // a normal variable, effective type uint32\_t, and this type never changes  uint32\_t a = 0.0;  // effective type of \*pa is uint32\_t, too, simply  // because \*pa is the object a  uint32\_t\* pa = &a;  // the object pointed to by q has no effective type, yet  void\* q = malloc(sizeof *uint32\_t*);  // the object pointed to by q still has no effective type,  // because nobody has written to it  uint32\_t\* qb = q;  // \*qb now has effective type uint32\_t because a uint32\_t value was written  \*qb = 37;  // the object pointed to by r has no effective type, yet, although  // it is initialized  void\* r = calloc(1, sizeof *uint32\_t*);  // the object pointed to by r still has no effective type,  // because nobody has written to or read from it  uint32\_t\* rc = r;  // \*rc now has effective type uint32\_t because a value is read  // from it with that type. The read operation is valid because we used calloc.  // Now the object pointed to by r (which is the same as \*rc) has  // gained an effective type, although we didn't change its value.  uint32\_t c = \*rc;  // the object pointed to by s has no effective type, yet.  void\* s = malloc(sizeof *uint32\_t*);  // the object pointed to by s now has effective type uint32\_t  // because an uint32\_t value is copied into it.  memcpy(*s*, *r*, sizeof *uint32\_t*); |

Lưu ý rằng với ví dụ cuối cùng, không cần thiết phải có một con trỏ uint32\_t\* đến đối tượng đó. Việc sao chép một đối tượng uint32\_t khác là đủ để có effective type

## RESTRICT QUALIFICATION

Nếu chúng ta có hai đối số con trỏ cùng kiểu dữ liệu, trình biên dịch không thể đưa ra bất kỳ giả định nào và sẽ luôn phải giả định rằng thay đổi \*e có thể thay đổi \*f:

|  |
| --- |
| void fun(float\* *e*, float\* *f*) {  float a = \**f*;  \**e* = 22;  float b = \**f*;  printf("is %g equal to %g?\n", a, b);  }  float fval = 4;  float eval = 77;  fun(&*eval*, &*fval*); |

tất cả diễn ra tốt đẹp và một cái gì đó giống như

|  |
| --- |
| is 4 equal to 4? |

được in. Nếu chúng ta truyền cùng một con trỏ, chương trình vẫn sẽ làm đúng và in

|  |
| --- |
| is 4 equal to 22? |

Điều này có thể trở nên không hiệu quả, nếu chúng ta biết bởi một số thông tin bên ngoài e và f sẽ không bao giờ trỏ đến cùng một đối tượng dữ liệu. Chúng ta có thể phản ánh kiến ​​thức đó bằng cách thêm từ khóa **restrict** vào các tham số con trỏ:

|  |
| --- |
| void fan(float\* restrict *e*, float\* restrict *f*) {  float a = \**f*;  \**e* = 22;  float b = \**f*;  printf("is %g equal to %g?\n", a, b);  } |

Sau đó, trình biên dịch có thể luôn cho rằng điều đó e và f trỏ đến các đối tượng khác nhau.

## THAY ĐỔI CÁC BYTE

Sau khi một đối tượng có một effective type, bạn không nên cố gắng sửa đổi nó thông qua một con trỏ thuộc kiểu khác, trừ khi kiểu khác đó là một kiểu ký tự, **char, signed char hoặc unsigned char.**

|  |
| --- |
| #include <inttypes.h>  #include <stdio.h>  int main(void) {  *uint32\_t* a = 57;  // Ép kiểu cho việc chuyển đổi từ các kiểu không tương thích!  unsigned char\* ap = (unsigned char\*)&a;  for (*size\_t* i = 0; i < sizeof a; ++i) {  /\* set each byte of a to 42 \*/  ap[i] = 42;  }  printf("a now has value %" PRIu32 "\n", a);  } |

Đây là một chương trình hợp lệ in ra:

|  |
| --- |
| a now has value 707406378 |

Cách hoạt động của mã code này:

* Truy cập được thực hiện vào các byte riêng lẻ của biến a được xem bằng kiểu unsigned char, vì vậy mỗi việc sửa đổi là xác định.
* Hai cách xem đối tượng, thông qua a và thông qua \*ap, gọi là alias, nhưng vì ap là con trỏ trỏ tới kiểu ký tự, quy tắc aliasing nghiêm ngặt không áp dụng. Do đó, trình biên dịch phải giả định rằng giá trị của a có thể đã thay đổi trong vòng lặp for. Giá trị đã sửa đổi của a phải được xây dựng từ các byte đã được thay đổi.
* Kiểu của a, uint32\_t không có các bit dư thừa (padding bits). Tất cả các bit của biểu diễn đều tính cho giá trị, ở đây là 707406378, và không thể có giá trị trap(bẫy).

## CÁC KIỂU KÍ TỰ KHÔNG THỂ TRUY CẬP THÔNG QUA CÁC KIỂU KHÔNG PHẢI KIỂU KÍ TỰ

Nếu một đối tượng được định nghĩa với kiểu lưu trữ tĩnh (static), thread (thread\_local), hoặc tự động (automatic), và nó có kiểu ký tự, bao gồm: char, unsigned char hoặc signed char, thì nó không được truy cập bằng một kiểu không phải ký tự. Trong ví dụ dưới đây, một mảng char được diễn giải dạng kiểu int, và hành vi của con trỏ int b không xác định khi truy cập các giá trị của mảng a.

|  |
| --- |
| #include <stdio.h>  int main(void) {  char a[100];  int\* b = (int\*)&a;  \*b = 1;  static char c[100];  b = (int\*)&c;  \*b = 2;  \_Thread\_local char d[100];  b = (int\*)&d;  \*b = 3;  } |

Đoạn mã này không xác định vì nó vi phạm quy tắc "kiểu hiệu quả" (effective type), không có đối tượng dữ liệu nào có kiểu hiệu quả có thể được truy cập thông qua một kiểu khác không phải ký tự. Vì kiểu khác ở đây là int, điều này không được phép.

Thậm chí nếu việc căn chỉnh (alignment) và kích thước con trỏ có thể phù hợp, điều này cũng không miễn trừ khỏi quy tắc này, hành vi vẫn là không xác định.

Điều này có nghĩa đặc biệt là không có cách nào trong C tiêu chuẩn để dành một đối tượng buffer với kiểu ký tự mà có thể sử dụng thông qua các con trỏ với các kiểu khác nhau, như bạn sử dụng một buffer được nhận từ malloc hoặc các hàm tương tự.

Một cách đúng để đạt được cùng mục tiêu như trong ví dụ trên là sử dụng union.

|  |
| --- |
| #include <stdio.h>  typedef union *bufType* *bufType*;  union *bufType* {  char c[sizeof(int[25])];  int i[25];  };  int main(void) {  *bufType* a = { .c = { 0 } }; // dành một buffer và khởi tạo  int\* b = a.i; // không cần ép kiểu  \*b = 1;  static *bufType* c = { .c = { 0 } };  b = c.i;  \*b = 2;  \_Thread\_local *bufType* d = { .c = { 0 } };  b = d.i;  \*b = 3;  } |

Ở đây, union đảm bảo rằng trình biên dịch biết từ đầu rằng buffer có thể được truy cập thông qua các view khác nhau. Điều này cũng có lợi thế là giờ đây buffer có một "view" là a.i và đã là kiểu int nên không cần phải chuyển đổi con trỏ.

## VI PHẠM QUY TẮC “STRICT ALIASING”

Trong đoạn mã sau, giả định đơn giản là kiểu dữ liệu float và uint32\_t có cùng kích thước.

|  |
| --- |
| #include <stdio.h>  #include <stdint.h>  void fun(*uint32\_t*\* *u*, float\* *f*) {  float a = \**f*;  \**u* = 22;  float b = \**f*;  printf("%g should equal %g\n", a, b);  } |

Biến u và f có kiểu cơ sở khác nhau, do đó trình biên dịch có thể giả định rằng chúng trỏ đến các đối tượng khác nhau. Không có khả năng là \*f có thể thay đổi giữa hai lần khởi tạo a và b, và do đó trình biên dịch có thể tối ưu mã thành một cái gì đó tương đương với:

|  |
| --- |
| void fun(*uint32\_t*\* *u*, float\* *f*) {  float a = \**f*;  \**u* = 22;  printf("%g should equal %g\n", a, a);  } |

Tức là, thao tác tải thứ hai của \*f có thể được tối ưu hoàn toàn.

Nếu chúng ta gọi hàm này "bình thường":

|  |
| --- |
| float fval = 4;  *uint32\_t* uval = 77;  fun(&uval, &fval); |

Tất cả đều ổn và một cái gì đó tương tự như:

|  |
| --- |
| 4 should equal 4 |

được in ra. Nhưng nếu chúng ta "lừa đảo" và truyền cùng một con trỏ, sau khi chuyển đổi nó:

|  |
| --- |
| float fval = 4;  *uint32\_t*\* up = (*uint32\_t*\*)&fval;  fun(up, &fval); |

Chúng ta vi phạm quy tắc aliasing nghiêm ngặt. Sau đó, hành vi trở thành không xác định. Kết quả có thể như trên, nếu trình biên dịch tối ưu hóa thao tác truy cập thứ hai, hoặc hoàn toàn khác nhau, và do đó chương trình của bạn sẽ trở thành một trạng thái không đáng tin cậy hoàn toàn.

# BIÊN DỊCH

Ngôn ngữ C truyền thống là một ngôn ngữ được biên dịch (trái ngược phiên dịch). Tiêu chuẩn C xác định các giai đoạn dịch, và kết quả của việc áp dụng chúng là một hình ảnh chương trình (hoặc chương trình đã được biên dịch). Trong phiên bản C11, các giai đoạn được liệt kê trong §5.1.1.2.

## TRÌNH BIÊN DỊCH (THE COMPILER)

Sau khi trình **tiền xử lý** C đã bao gồm tất cả các **tệp tiêu đề** và mở rộng tất cả **các macro**, trình biên dịch có thể biên dịch chương trình. Nó làm điều này bằng cách chuyển mã nguồn C thành một tệp **mã đối tượng**, đó là một tệp có đuôi .o chứa phiên bản nhị phân của mã nguồn. Tuy nhiên mã đối tượng không thể thực thi trực tiếp. Để tạo một chương trình thực thi, bạn cũng phải thêm mã cho tất cả các hàm thư viện đã được #include vào tệp (điều này không giống như việc bao gồm các khai báo, đó là điều mà #include làm). Điều này là công việc của trình liên kết (**Linker**).

Nói chung, trình tự chính xác cách gọi trình biên dịch C phụ thuộc nhiều vào hệ thống mà bạn đang sử dụng. Ở đây chúng tôi đang sử dụng trình biên dịch GCC, mặc dù cần lưu ý rằng còn tồn tại nhiều trình biên dịch khác:

|  |
| --- |
| % gcc -Wall -c foo.c |

‘%’ là dấu nhắc lệnh của hệ điều hành. Dòng lệnh này yêu cầu trình biên dịch chạy trước tiên trình tiền xử lý trên tệp **foo.c** sau đó biên dịch nó thành tệp mã đối tượng **foo.o**. Tùy chọn **-c** có nghĩa là biên dịch tệp mã nguồn thành tệp đối tượng nhưng không gọi trình liên kết. Tùy chọn **-c** này có sẵn trên các hệ thống POSIX, như Linux hoặc macOS; các hệ thống khác có thể sử dụng cú pháp khác nhau.

Nếu toàn bộ chương trình của bạn nằm trong một tệp mã nguồn, bạn có thể làm như sau:

|  |
| --- |
| % gcc -Wall foo.c -o foo |

Dòng lệnh này yêu cầu trình biên dịch chạy trước tiên trình tiền xử lý trên foo.c, sau đó biên dịch nó và liên kết nó để tạo một chương trình có tên là foo. Tùy chọn -o cho biết rằng từ tiếp theo trên dòng là tên tệp thực thi nhị phân (chương trình). Nếu bạn không chỉ định -o (nếu bạn chỉ gõ gcc foo.c), chương trình thực thi sẽ có tên a.out vì lý do lịch sử.

Nói chung, trình biên dịch thực hiện bốn bước khi chuyển đổi một tệp .c thành một chương trình thực thi:

1. **Tiền xử lý (pre-processing)** - mở rộng văn bản các chỉ thị #include và macro #define trong tệp .c của bạn.
2. **Biên dịch (compilation)** - chuyển đổi chương trình thành mã hợp ngữ (bạn có thể dừng trình biên dịch ở bước này bằng cách thêm tùy chọn -S).
3. **Hợp ngữ (assembly)** - chuyển đổi mã hợp ngữ thành mã máy.
4. **Liên kết (linkage**) - liên kết mã đối tượng với các thư viện bên ngoài để tạo một chương trình thực thi.

Lưu ý rằng tên trình biên dịch chúng ta đang sử dụng là GCC, viết tắt của cả "GNU C compiler" và "GNU compiler collection", phụ thuộc vào ngữ cảnh. Có các trình biên dịch C khác tồn tại. Đối với các hệ điều hành giống Unix, nhiều trong số chúng có tên cc, viết tắt của "C compiler", thường là liên kết mục tiêu cho một số trình biên dịch khác. Trên các hệ thống Linux, cc thường là một bí danh của GCC. Trên macOS hoặc OS-X, nó trỏ đến clang.

Tiêu chuẩn POSIX hiện tại mục đích c99 là tên của trình biên dịch C — nó hỗ trợ tiêu chuẩn C99 theo mặc định. Trước đó, phiên bản POSIX mục đích là trình biên dịch c89. POSIX cũng mục đích rằng trình biên dịch này hiểu các tùy chọn -c và -o mà chúng ta đã sử dụng ở trên.

Lưu ý: Tùy chọn -Wall xuất hiện trong cả hai ví dụ gcc trên để yêu cầu trình biên dịch in ra cảnh báo về các cấu trúc đáng ngờ, điều này được đề xuất mạnh mẽ. Nên thêm các tùy chọn cảnh báo khác, ví dụ: -Wextra.

## FILE TYPES

Biên dịch chương trình C yêu cầu bạn làm việc với năm loại tệp (files):

1. **Tệp nguồn (Source files):** Đây là các tệp chứa định nghĩa hàm và thường có đuôi .c theo quy ước. Lưu ý rằng .cc và .cpp là tệp C++, không phải tệp C.

Ví dụ: foo.c

1. **Tệp tiêu đề (Header files):** Đây là các tệp chứa các khai báo hàm và các câu lệnh tiền xử lý khác (xem bên dưới). Chúng được sử dụng để cho phép các tệp mã nguồn truy cập vào các hàm được định nghĩa bên ngoài. Tệp tiêu đề kết thúc bằng .h theo quy ước.

Ví dụ: foo.h

1. **Tệp đối tượng (Object files):** Đây là các tệp được tạo ra như kết quả của trình biên dịch. Chúng bao gồm định nghĩa hàm dưới dạng nhị phân, nhưng chúng không thể thực thi một mình. Tệp đối tượng kết thúc bằng .o theo quy ước, tuy nhiên trên một số hệ điều hành (ví dụ: Windows, MS-DOS), chúng thường kết thúc bằng .obj.

Ví dụ: foo.o foo.obj

1. **Tệp thực thi nhị phân (Binary executables):** Đây là kết quả của trình liên kết. Trình liên kết liên kết một số tệp đối tượng để tạo ra một tệp nhị phân có thể thực thi trực tiếp. Trên hệ điều hành Unix, tệp thực thi nhị phân không có đuôi đặc biệt, tuy nhiên chúng thường kết thúc bằng .exe trên Windows.

Ví dụ: foo foo.exe

1. **Thư viện (Libraries):** Một thư viện là một tệp nhị phân được biên dịch, nhưng không phải là một tệp thực thi (tức không có hàm main() trong thư viện). Một thư viện chứa các hàm có thể được sử dụng bởi nhiều chương trình. Thư viện nên đi kèm với các tệp tiêu đề chứa các khai báo cho tất cả các hàm trong thư viện; các tệp tiêu đề này nên được tham chiếu (ví dụ: #include <library.h>) trong bất kỳ tệp mã nguồn nào sử dụng thư viện. Sau đó, trình liên kết cần tham chiếu đến thư viện để chương trình có thể được biên dịch thành công. Có hai loại thư viện: static và dynamic.

* **Thư viện tĩnh (Static library):** Một thư viện tĩnh (.a trên các hệ thống POSIX và .lib trên Windows — không nên nhầm lẫn với các tệp thư viện nhập khẩu DLL, cũng sử dụng phần mở rộng .lib) được xây dựng tĩnh vào chương trình. Thư viện tĩnh có lợi thế là chương trình biết chính xác phiên bản của thư viện nào được sử dụng. Tuy nhiên, kích thước của các tệp thực thi sẽ lớn hơn vì tất cả các hàm thư viện được sử dụng đều được bao gồm.

Ví dụ: libfoo.a foo.lib

* **Thư viện động (Dynamic library):** Một thư viện động (.so trên hầu hết các hệ thống POSIX, .dylib trên OSX và .dll trên Windows) được liên kết động vào thời gian chạy bởi chương trình. Đôi khi còn được gọi là thư viện chia sẻ vì một hình ảnh thư viện có thể được chia sẻ bởi nhiều chương trình. Thư viện động có lợi thế tiết kiệm không gian đĩa nếu nhiều ứng dụng sử dụng cùng một thư viện. Hơn nữa, chúng cho phép cập nhật thư viện (sửa lỗi) mà không cần phải xây dựng lại các tệp thực thi.

Ví dụ: foo.so foo.dylib foo.dll

## THE LINKER (TRÌNH LIÊN KẾT)

Trình liên kết có nhiệm vụ liên kết các tệp đối tượng (.o files) thành một chương trình thực thi nhị phân. Quá trình liên kết chủ yếu bao gồm giải quyết các địa chỉ tượng trưng thành các địa chỉ số. Kết quả của quá trình liên kết thường là một chương trình thực thi.

Trong quá trình liên kết, trình liên kết sẽ thu thập tất cả các mô-đun đối tượng được chỉ định trên dòng lệnh, thêm vào mã khởi đầu cụ thể của hệ thống và cố gắng giải quyết tất cả các tham chiếu ngoài trong các mô-đun đối tượng với các định nghĩa bên ngoài trong các tệp đối tượng khác (các tệp đối tượng có thể được chỉ định trực tiếp trên dòng lệnh hoặc được thêm vào ngầm qua thư viện). Sau đó, trình liên kết sẽ gán địa chỉ nạp cho các tệp đối tượng, tức là, nó chỉ định nơi mã và dữ liệu sẽ xuất hiện trong không gian địa chỉ của chương trình hoàn chỉnh. Khi đã có địa chỉ nạp, nó có thể thay thế tất cả các địa chỉ tượng trưng trong mã đối tượng bằng địa chỉ "thực", số học trong không gian địa chỉ của mục tiêu. Chương trình đã sẵn sàng để thực thi.

Quá trình này bao gồm cả các tệp đối tượng được trình biên dịch từ các tệp mã nguồn của bạn cũng như các tệp đối tượng đã được biên dịch trước cho bạn và được tập hợp thành các tệp thư viện. Các tệp này có tên kết thúc bằng .a hoặc .so, và bạn thường không cần phải biết về chúng, vì trình liên kết biết vị trí của hầu hết chúng và sẽ liên kết chúng vào tự động khi cần thiết.

### Liên kết ngầm định của trình liên kết

Giống như trình tiền xử lý, trình liên kết là một chương trình riêng biệt, thường được gọi là ld (nhưng Linux sử dụng collect2, ví dụ). Giống như trình tiền xử lý, trình liên kết được gọi tự động cho bạn khi bạn sử dụng trình biên dịch. Do đó, cách thông thường để sử dụng trình liên kết là như sau:

|  |
| --- |
| % gcc foo.o bar.o baz.o -o myprog |

Dòng lệnh này yêu cầu trình biên dịch liên kết ba tệp đối tượng (foo.o, bar.o và baz.o) thành một tệp nhị phân thực thi có tên myprog. Bây giờ bạn có một tệp có tên myprog mà bạn có thể chạy và có hy vọng sẽ thực hiện điều gì đó thú vị và/hoặc hữu ích.

### Yêu cầu rõ ràng của trình liên kết

Có thể gọi trình liên kết trực tiếp, nhưng điều này thường không được khuyến khích và thường đặc thù cho mỗi nền tảng. Điều này có nghĩa là các tùy chọn hoạt động trên Linux không nhất thiết phải hoạt động trên Solaris, AIX, macOS, Windows và tương tự đối với bất kỳ nền tảng nào khác. Nếu bạn làm việc với GCC, bạn có thể sử dụng gcc -v để xem những gì được thực thi thay cho bạn.

### Tùy chọn cho trình liên kết

Trình liên kết cũng có một số đối số để thay đổi hành vi của nó. Lệnh sau sẽ yêu cầu gcc liên kết foo.o và bar.o, nhưng cũng bao gồm thư viện ncurses.

|  |
| --- |
| % gcc foo.o bar.o -o foo -lncurses |

Điều này thực sự (hơn hoặc ít hơn) tương đương với

|  |
| --- |
| % gcc foo.o bar.o /usr/lib/libncurses.so -o foo |

(tuy nhiên libncurses.so có thể là libncurses.a, đó chỉ là một lưu trữ được tạo ra với ar). Lưu ý rằng bạn nên liệt kê các thư viện (hoặc thông qua các tùy chọn -lname) sau các tệp đối tượng. Với thư viện tĩnh, thứ tự mà chúng được chỉ định quan trọng; thường, với các thư viện chia sẻ, thứ tự không quan trọng.

Lưu ý rằng trên nhiều hệ thống, nếu bạn sử dụng các hàm toán học (từ <math.h>), bạn cần chỉ định -lm để tải thư viện toán học - nhưng macOS và macOS Sierra không yêu cầu điều này. Có các thư viện khác là các thư viện riêng biệt trên Linux và các hệ thống Unix khác, nhưng không phải trên macOS - các tiểu chuẩn POSIX, và POSIX realtime, và các thư viện mạng là ví dụ. Do đó, quá trình liên kết thay đổi giữa các nền tảng.

### Những tùy chọn biên dịch khác

Điều này là tất cả những gì bạn cần biết để bắt đầu biên dịch chương trình C của riêng bạn. Nói chung, chúng tôi cũng khuyến nghị bạn sử dụng tùy chọn dòng lệnh -Wall:

|  |
| --- |
| % gcc -Wall -c foo.c |

Tùy chọn -Wall cho phép trình biên dịch cảnh báo về các cấu trúc mã hợp pháp nhưng đáng nghi và sẽ giúp bạn phát hiện rất nhiều lỗi sớm.

Nếu bạn muốn trình biên dịch ném nhiều cảnh báo hơn (bao gồm các biến được khai báo nhưng không được sử dụng, quên trả về một giá trị v.v.), bạn có thể sử dụng tập lệnh này, vì -Wall, bất kể tên gọi, không bật tất cả các cảnh báo có thể có:

|  |
| --- |
| % gcc -Wall -Wextra -Wfloat-equal -Wundef -Wcast-align -Wwrite-strings -Wlogical-op \  > -Wmissing-declarations -Wredundant-decls -Wshadow … |

Lưu ý rằng clang có tùy chọn -Weverything thực sự bật tất cả các cảnh báo trong clang.

## TIỀN XỬ LÝ (THE PREPROCESSOR)

Trước khi trình biên dịch C bắt đầu biên dịch một tệp mã nguồn, tệp đó được xử lý trong một giai đoạn tiền xử lý. Giai đoạn này có thể được thực hiện bởi một chương trình riêng biệt hoặc hoàn toàn tích hợp trong một tệp thực thi duy nhất. Dù cho trường hợp nào, nó được tự động gọi bởi trình biên dịch trước khi bắt đầu quá trình biên dịch chính thức. Giai đoạn tiền xử lý chuyển đổi mã nguồn của bạn thành mã nguồn hoặc đơn vị dịch thuật khác bằng cách áp dụng các thay thế văn bản. Bạn có thể xem nó như là một mã nguồn "sửa đổi" hoặc "mở rộng". Mã nguồn được mở rộng này có thể tồn tại dưới dạng tệp thực tế trong hệ thống tệp hoặc nó có thể chỉ được lưu trữ trong bộ nhớ trong một khoảng thời gian ngắn trước khi được xử lý tiếp.

Các lệnh tiền xử lý bắt đầu bằng dấu thăng (#). Có một số lệnh tiền xử lý; hai lệnh quan trọng nhất là:

1. **Defines:**

#define chủ yếu được sử dụng để định nghĩa các hằng số. Ví dụ:

|  |
| --- |
| #define BIGNUM 1000000  int a = BIGNUM; |

Sẽ trở thành

|  |
| --- |
| int a = 1000000; |

#define được sử dụng theo cách này để tránh việc phải viết rõ ràng một số hằng số trong nhiều vị trí khác nhau trong tệp mã nguồn. Điều này quan trọng trong trường hợp bạn cần thay đổi giá trị hằng số sau này; việc thay đổi nó một lần trong #define ít gặp lỗi hơn việc phải thay đổi nó ở nhiều vị trí khác nhau trên mã.

Vì #define chỉ thực hiện tìm kiếm và thay thế nâng cao, bạn cũng có thể khai báo các macros. Ví dụ:

|  |
| --- |
| #define ISTRUE(*stm*) do{stm = stm ? 1 : 0;}while(0)  // trong hàm:  a = x;  ISTRUE(a); |

Sẽ trở thành

|  |
| --- |
| // trong hàm:  a = x;  do {  a = a ? 1 : 0;  } while(0); |

Ở phép tính gần đúng đầu tiên, hiệu ứng này là tương tự như với hàm inline, nhưng trình tiền xử lý không cung cấp kiểm tra kiểu cho các macros #define. Điều này đã được biết đến là dễ gây lỗi và việc sử dụng chúng đòi hỏi cẩn thận.

Lưu ý ở đây, trình tiền xử lý cũng sẽ thay thế các chú thích bằng khoảng trắng như đã được giải thích ở dưới.

1. **Includes:**

|  |
| --- |
| #include được sử dụng để truy cập các định nghĩa hàm được xác định bên ngoài một tệp mã nguồn. Ví dụ:  #include <stdio.h> |

sẽ khiến trình tiền xử lý dán nội dung của <stdio.h> vào tệp mã nguồn tại vị trí câu lệnh #include trước khi được biên dịch. #include hầu như luôn được sử dụng để bao gồm các tệp tiêu đề, các tệp chứa chủ yếu các khai báo hàm và các câu lệnh #define. Trong trường hợp này, chúng ta sử dụng #include để có thể sử dụng các hàm như printf và scanf, các khai báo của chúng được đặt trong tệp stdio.h. Trình biên dịch C không cho phép bạn sử dụng một hàm trừ khi nó đã được khai báo hoặc định nghĩa trước trong tệp đó; câu lệnh #include là cách để tái sử dụng mã đã được viết trước đó trong chương trình C của bạn.

1. **Các phép toán Logic**

|  |
| --- |
| #if defined A || defined B  variable = another\_variable + 1;  #else  variable = another\_variable \* 2;  #endif |

Sẽ được thay đổi thành:

|  |
| --- |
| variable = another\_variable + 1; |

nếu A hoặc B được xác định ở đâu đó trong dự án trước đó. Nếu điều này không đúng, tất nhiên trình tiền xử lý sẽ thực hiện điều này:

|  |
| --- |
| variable = another\_variable \* 2; |

Điều này thường được sử dụng cho mã chạy trên các hệ thống khác nhau hoặc biên dịch trên các trình biên dịch khác nhau. Vì có các định nghĩa toàn cầu, được xác định cho trình biên dịch / hệ thống cụ thể, bạn có thể thử nghiệm trên những định nghĩa đó và luôn để trình biên dịch chỉ sử dụng mã mà anh ta sẽ biên dịch chắc chắn.

1. **Comments**

Trình tiền xử lý thay thế tất cả các chú thích trong tệp nguồn bằng một khoảng trắng duy nhất. Chú thích được chỉ định bằng // cho đến cuối dòng hoặc một kết hợp dấu mở /\* và dấu đóng \*/.

## THE TRANSLATION PHASES (CÁC GIAI ĐOẠN DỊCH)

Tiêu chuẩn C 2011 xác định quá trình dịch mã nguồn thành hình ảnh chương trình (ví dụ, tệp thực thi) trong 8 bước theo thứ tự, được liệt kê trong §5.1.1.2 Translation Phases:

1. Dữ liệu đầu vào của tệp nguồn được ánh xạ vào tập ký tự nguồn (nếu cần). Trigraphs, là các chuỗi ba ký tự được sử dụng để biểu diễn các ký tự đặc biệt trong các bộ ký tự không có các ký tự này, sẽ được thay thế ở bước này.
2. Các dòng tiếp tục, là các dòng kết thúc bằng dấu gạch chéo ngược (), được ghép với dòng kế tiếp, hiệu quả tạo ra một dòng logic từ nhiều dòng vật lý.
3. Mã nguồn được phân tích thành các token trắng và tiền xử lý. Các khoảng trống, bình luận và các ký tự phân tách khác được sử dụng để tách mã nguồn thành các token riêng lẻ, chẳng hạn như từ khóa, nhận dạng, toán tử và hằng số.
4. Tiền xử lý được áp dụng, thực thi các chỉ thị, mở rộng các macro và áp dụng các pragma. Tiền xử lý xử lý mã nguồn, thay thế các macro bằng các định nghĩa tương ứng, xử lý các chỉ thị #include để bao gồm các tệp khác và thực thi các chỉ thị tiền xử lý khác. Mỗi tệp nguồn được đưa vào bằng #include trải qua các bước dịch 1 đến 4 đệ quy nếu cần. Tất cả các chỉ thị liên quan đến tiền xử lý sau đó được xóa khỏi mã đã xử lý.
5. Giá trị tập ký tự nguồn trong các hằng số ký tự và chuỗi được ánh xạ vào tập ký tự thực thi. Bước này đảm bảo rằng các hằng số ký tự và chuỗi được biểu diễn bằng cách sử dụng mã hóa phù hợp cho môi trường đích.
6. Chuỗi ký tự liền kề với nhau sẽ được nối lại. Nếu có nhiều chuỗi ký tự được đặt liên tiếp nhau, trình biên dịch kết hợp chúng thành một chuỗi ký tự duy nhất.
7. Mã nguồn được phân tích thành các token, tạo thành đơn vị dịch. Các token được phân loại tiếp theo dựa trên loại của chúng, chẳng hạn như nhận dạng, từ khóa, toán tử và hằng số.
8. Các tham chiếu bên ngoài được giải quyết và hình ảnh chương trình được hình thành. Trong bước này, trình biên dịch liên kết các đơn vị dịch khác nhau, giải quyết các tham chiếu bên ngoài đến các hàm và biến và tạo ra hình ảnh chương trình cuối cùng hoặc tệp thực thi.

Lưu ý rằng một triển khai của trình biên dịch C có thể kết hợp nhiều bước lại với nhau để tăng hiệu suất, miễn là chương trình kết quả vẫn hoạt động như nếu các bước trên đã xảy ra riêng biệt và theo thứ tự đã chỉ định.

# INLINE ASSEMBLY

## GCC INLINE ASSEMBLY IN MACROS

Chúng ta có thể đặt các lệnh hợp ngữ bên trong macro và sử dụng macro giống như cách bạn gọi một hàm.

|  |
| --- |
| #define mov(*x*,*y*) \  { \  \_\_asm\_\_ ("l.cmov %0,%1,%2" : "=r" (x) : "r" (y), "r" (0x0000000F)); \  }  // một số định nghĩa và gán  unsigned char sbox[size][size];  unsigned char sbox[size][size];  // Sử dụng  mov(state[0][1], sbox[si][sj]); |

Sử dụng các lệnh trình biên dịch hợp ngữ nhúng trong mã C có thể cải thiện thời gian chạy của chương trình. Điều này rất hữu ích trong các tình huống thời gian quan trọng như các thuật toán mật mã như AES. Ví dụ: đối với một thao tác dịch đơn giản cần thiết trong thuật toán AES, chúng ta có thể thay thế một lệnh trình biên dịch hợp ngữ xoay phải trực tiếp bằng toán tử dịch C >>.

Trong một triển khai của 'AES256', trong hàm 'AddRoundKey()', chúng ta có một số câu như sau:

|  |
| --- |
| unsigned int w; // 32-bit  unsigned char subkey[4]; // 8-bit, 4\*8 = 32  subkey[0] = w >> 24; // giữ 8 bit, MSB, nhóm 8 bit ngoài cùng bên trái  subkey[1] = w >> 16; // giữ 8 bit, nhóm 8 bit thứ hai từ bên trái  subkey[2] = w >> 8; // giữ 8 bit, nhóm 8 bit thứ hai từ bên phải  subkey[3] = w; // giữ 8 bit, LSB, nhóm 8 bit ngoài cùng bên phải  /// subkey <- w |

Chúng đơn giản gán giá trị bit của w cho mảng subkey.

Chúng ta có thể thay đổi ba biểu thức C dịch + gán và một biểu thức gán với chỉ một thao tác xoay phải trình biên dịch hợp ngữ.

|  |
| --- |
| \_\_asm\_\_ ("l.ror %0,%1,%2" : "=r" (\* (unsigned int \*) subkey) : "r" (w), "r" (0x10)); |

Kết quả cuối cùng chính xác như nhau.

## HỖ TRỢ ASM CƠ BẢN TRONG GCC

Hỗ trợ asm cơ bản với gcc có cú pháp như sau:

|  |
| --- |
| asm [ volatile ] ( AssemblerInstructions ) |

Trong đó, AssemblerInstructions là mã assembly trực tiếp cho bộ xử lý được sử dụng. Từ khóa volatile là tùy chọn và không có tác dụng gì vì gcc không tối ưu hóa mã trong một câu lệnh asm cơ bản. AssemblerInstructions có thể chứa nhiều lệnh assembly. Một câu lệnh asm cơ bản được sử dụng khi bạn có một hàm asm phải tồn tại bên ngoài một hàm C. Ví dụ sau đây được lấy từ tài liệu GCC:

|  |
| --- |
| /\* Lưu ý rằng đoạn mã này sẽ không biên dịch được với tùy chọn -masm=intel \*/  #define DebugBreak() asm("int $3") |

Trong ví dụ này, bạn có thể sử dụng DebugBreak() ở nơi khác trong mã của bạn và nó sẽ thực thi lệnh assembly "int $3". Lưu ý rằng mặc dù gcc sẽ không thay đổi bất kỳ mã nào trong một câu lệnh asm cơ bản, trình tối ưu hóa vẫn có thể di chuyển các câu lệnh asm liền nhau. Nếu bạn có nhiều lệnh assembly phải xảy ra theo một thứ tự cụ thể, hãy bao gồm chúng trong một câu lệnh asm duy nhất.

## HỖ TRỢ ASM MỞ RỘNG TRONG GCC

Hỗ trợ asm mở rộng trong gcc có cú pháp như sau:

|  |
| --- |
| asm [volatile] ( AssemblerTemplate  : OutputOperands  [ : InputOperands  [ : Clobbers ] ])  asm [volatile] goto ( AssemblerTemplate  :  : InputOperands  : Clobbers  : GotoLabels) |

Trong đó, AssemblerTemplate là mẫu cho lệnh assembler, OutputOperands là bất kỳ biến C nào có thể bị thay đổi bởi mã assembly, InputOperands là bất kỳ biến C nào được sử dụng như tham số đầu vào, Clobbers là danh sách các thanh ghi bị thay đổi bởi mã assembly, và GotoLabels là bất kỳ nhãn lệnh goto nào có thể được sử dụng trong mã assembly.

Định dạng mở rộng này được sử dụng trong các hàm C và là cách sử dụng inline assembly phổ biến hơn. Dưới đây là một ví dụ từ nhân kernel Linux để hoán đổi byte của các số 16-bit và 32-bit trên bộ xử lý ARM:

|  |
| --- |
| /\* Từ arch/arm/include/asm/swab.h trong phiên bản kernel Linux 4.6.4 \*/  #if \_\_LINUX\_ARM\_ARCH\_\_ >= 6  static inline \_\_attribute\_const\_\_ \_\_u32 \_\_arch\_swahb32(\_\_u32 *x*)  {  \_\_asm\_\_ ("rev16 %0, %1" : "=r" (x) : "r" (x));  return x;  }  #define \_\_arch\_swahb32 \_\_arch\_swahb32  #define \_\_arch\_swab16(*x*) ((\_\_u16)\_\_arch\_swahb32(x))  static inline \_\_attribute\_const\_\_ \_\_u32 \_\_arch\_swab32(\_\_u32 *x*)  {  \_\_asm\_\_ ("rev %0, %1" : "=r" (x) : "r" (x));  return x;  }  #define \_\_arch\_swab32 \_\_arch\_swab32  #endif |

Mỗi phần asm sử dụng biến x làm tham số đầu vào và đầu ra của nó. Hàm C sau đó trả về kết quả được thay đổi.

Với định dạng asm mở rộng, gcc có thể tối ưu các lệnh assembly trong khối asm theo các quy tắc tối ưu hóa mã C. Nếu bạn muốn khối asm của bạn không bị tác động, hãy sử dụng từ khóa volatile cho khối asm.

# PHẠM VI CỦA ĐỊNH DANH

## Phạm vi của khai báo hàm

Khi chúng ta khai báo một hàm trong ngôn ngữ C, tên của các tham số trong khai báo hàm đó chỉ có phạm vi tồn tại bên trong khai báo hàm và không có ý nghĩa bên ngoài nó. Dưới đây là một ví dụ minh họa:

|  |
| --- |
| #include <stdio.h>  int test\_function(int *apple*); // Khai báo hàm test\_function với tham số apple  int main(void)  {  int orange = 5; // Khởi tạo biến orange với giá trị 5  orange = test\_function(orange); // Gọi hàm test\_function và truyền biến orange vào làm tham số  printf("%d\r\n", orange); // In giá trị của biến orange sau khi gọi hàm (orange = 6)  return 0;  }  int test\_function(int *fruit*) // Định nghĩa hàm test\_function với tham số fruit  {  *fruit* += 1; // Tăng giá trị của tham số fruit lên 1  return *fruit*; // Trả về giá trị của tham số fruit  } |

Lưu ý rằng bạn sẽ nhận được thông báo lỗi khó hiểu nếu bạn giới thiệu tên một kiểu trong một khai báo hàm

|  |
| --- |
| int function(struct *whatever* \**arg*);  struct *whatever*  {  int a;  // ...  };  int function(struct *whatever* \**arg*)  {  return *arg*->a;  } |

Với GCC 6.3.0, mã nguồn này (tệp dc11.c) sẽ tạo ra thông báo lỗi:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c  dc11.c:1:25: error: ‘struct whatever’ declared inside parameter list will not be visible outside of this definition or declaration [-Werror]  int function(struct whatever \*arg);  ^~~~~~~~  dc11.c:9:9: error: conflicting types for ‘function’  int function(struct whatever \*arg)  ^~~~~~~~  dc11.c:1:9: note: previous declaration of ‘function’ was here  int function(struct whatever \*arg);  ^~~~~~~~  cc1: all warnings being treated as errors  $ |

Đặt định nghĩa structure trước khai báo hàm hoặc thêm "struct whatever;" trước khai báo hàm, và không có vấn đề gì xảy ra. Bạn không nên giới thiệu các tên kiểu mới trong khai báo hàm vì không có cách nào sử dụng kiểu đó và do đó không thể định nghĩa hoặc sử dụng hàm đó.

## Phạm vi khối

Một identifier có phạm vi khối nếu khai báo tương ứng của nó xuất hiện bên trong một khối (khai báo tham số trong định nghĩa hàm áp dụng). Phạm vi kết thúc ở cuối khối tương ứng.

Không có các thực thể khác nhau với cùng một identifier có thể có cùng phạm vi, nhưng phạm vi có thể trùng lặp. Trong trường hợp trùng lặp phạm vi, chỉ thực thể được khai báo trong phạm vi bên trong nhất mới hiển thị.

|  |
| --- |
| #include <stdio.h>  void test(int *bar*) // bar có phạm vi khối hàm test  {  int foo = 5; // foo có phạm vi khối hàm test  {  int bar = 10; // bar có phạm vi khối bên trong, điều này trùng lặp với khai báo test:bar trước đó và nó che khuất test:bar  printf("%d %d\n", foo, bar); // 5 10  } // kết thúc phạm vi cho thanh bên trong  printf("%d %d\n", foo, *bar*); // 5 5, ở đây bar là test:bar  } // kết thúc phạm vi cho test:foo và test:bar  int main(void)  {  int foo = 3; // foo có phạm vi khối hàm chính  printf("%d\n", foo); // 3  test(5);  printf("%d\n", foo); // 3  return 0;  } // kết thúc phạm vi cho main:fo |

## Phạm vi của tệp

|  |
| --- |
| #include <stdio.h>  /\* Định danh, foo, được khai báo bên ngoài tất cả các khối.  Nó có thể được sử dụng ở bất cứ đâu sau khi khai báo cho đến khi kết thúc bộ dịch.\*/  static int foo;  void test\_function(void)  {  foo += 2;  }  int main(void)  {  foo = 1;  test\_function();  printf("%d\r\n", foo); //foo = 3;  return 0;  } |

## Phạm vi hàm

Phạm vi hàm là phạm vi đặc biệt dành cho các nhãn (labels) trong ngôn ngữ lập trình C. Điều này do tính chất đặc biệt của nhãn. Một nhãn có thể được nhìn thấy trong toàn bộ hàm mà nó được định nghĩa, và ta có thể nhảy đến nó (sử dụng câu lệnh goto với tên nhãn) từ bất kỳ điểm nào trong cùng một hàm. Mặc dù không có ích lợi nhiều, ví dụ dưới đây minh họa điều này:

|  |
| --- |
| #include <stdio.h>  int main(int *argc*, char\* *argv*[]) {  int a = 0;  goto INSIDE;  OUTSIDE:  if (a != 0) {  int i = 0;  INSIDE:  printf("a=%d\n", a);  goto OUTSIDE;  }  } |

Nhãn INSIDE có vẻ như được định nghĩa trong khối if, giống như biến i mà phạm vi của nó chỉ trong khối. Tuy nhiên, thực tế là nhãn INSIDE có thể nhìn thấy trong toàn bộ hàm, như ví dụ về câu lệnh goto INSIDE; minh họa. Do đó, không thể có hai nhãn cùng tên trong một hàm duy nhất.

Một cách sử dụng tiềm năng của việc sử dụng nhãn như CLEANUP1 và CLEANUP2 trong ví dụ dưới đây là thực hiện các công việc dọn dẹp phức tạp cho các tài nguyên đã cấp phát:

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  void a\_function(void) {  double\* a = malloc(sizeof(double[34]));  if (!a) {  fprintf(stderr, "can't allocate\n");  return; /\* Không cần giải phóng a nếu nó là null \*/  }  *FILE*\* b = fopen("some\_file", "r");  if (!b) {  fprintf(stderr, "can't open\n");  goto CLEANUP1; /\* Giải phóng a; không cần đóng b \*/  }  /\* làm một số công việc hợp lý \*/  if (error) {  fprintf(stderr, "something's wrong\n");  goto CLEANUP2; /\* Giải phóng a và đóng b để tránh rò rỉ \*/  }  /\* làm thêm một số công việc khác \*/  CLEANUP2:  close(b);  CLEANUP1:  free(a);  } |

Nhãn như CLEANUP1 và CLEANUP2 là các định danh đặc biệt có cách hoạt động khác biệt so với tất cả các định danh khác. Chúng có thể nhìn thấy từ bất kỳ đâu trong hàm, ngay cả trong những nơi được thực thi trước câu lệnh có nhãn, hoặc ngay cả trong những nơi không bao giờ thực thi nếu không có lệnh goto nào được thực thi. Thường, nhãn được viết bằng chữ thường thay vì chữ in hoa.

# CHUYỂN ĐỔI RÕ RÀNG VÀ CHUYỂN ĐỔI NGẦM ĐỊNH

## CHUYỂN ĐỔI SỐ NGUYÊN TRONG LỜI GỌI HÀM

Trong ngôn ngữ lập trình C, khi gọi một hàm với tham số kiểu số nguyên, giá trị số nguyên sẽ được mở rộng (widened) theo quy tắc chuyển đổi số nguyên được định nghĩa trong tiêu chuẩn C11 6.3.1.3.

**6.3.1.3 - Số nguyên có dấu và không có dấu**

Khi giá trị có kiểu số nguyên được chuyển đổi sang kiểu số nguyên khác (trừ kiểu \_Bool), nếu giá trị có thể được biểu diễn bởi kiểu mới, nó không thay đổi.

Nếu kiểu mới là kiểu không dấu (unsigned), giá trị sẽ được chuyển đổi bằng cách lặp đi lặp lại việc cộng hoặc trừ một giá trị lớn hơn một lần giá trị tối đa có thể biểu diễn trong kiểu mới cho đến khi giá trị thuộc khoảng giá trị của kiểu mới.

Nếu kiểu mới là kiểu có dấu (signed) và giá trị không thể biểu diễn trong kiểu này, kết quả sẽ được xác định bởi cài đặt của trình biên dịch hoặc một tín hiệu đã xác định bởi cài đặt sẽ được kích hoạt.

Thường thì bạn không nên cắt giảm (truncate) một kiểu số nguyên lớn có dấu thành một kiểu số nguyên nhỏ hơn, vì rõ ràng các giá trị không thể vừa vặn và không có ý nghĩa rõ ràng cho điều này. Tiêu chuẩn C đã định nghĩa những trường hợp này là "được xác định bởi cài đặt", có nghĩa là chúng không di động (portable).

Ví dụ dưới đây giả định rằng int có độ rộng 32 bit:

|  |
| --- |
| #include <stdio.h>  #include <stdint.h>  void param\_u8(*uint8\_t* *val*) {  printf("%s val is %d\n", \_\_func\_\_, *val*); /\* val được nâng cấp thành int \*/  }  void param\_u16(*uint16\_t* *val*) {  printf("%s val is %d\n", \_\_func\_\_, *val*); /\* val được nâng cấp thành int \*/  }  void param\_u32(*uint32\_t* *val*) {  printf("%s val is %u\n", \_\_func\_\_, *val*); /\* ở đây val phù hợp với unsigned \*/  }  void param\_u64(*uint64\_t* *val*) {  printf("%s val is " PRI64u "\n", \_\_func\_\_, val); /\* Được cố định với chuỗi định dạng \*/  }  void param\_s8(*int8\_t* *val*) {  printf("%s val is %d\n", \_\_func\_\_, *val*); /\* val được nâng cấp thành int \*/  }  void param\_s16(*int16\_t* *val*) {  printf("%s val is %d\n", \_\_func\_\_, *val*); /\* val được nâng cấp thành int \*/  }  void param\_s32(*int32\_t* *val*) {  printf("%s val is %d\n", \_\_func\_\_, *val*); /\* val có cùng độ rộng như int \*/  }  void param\_s64(*int64\_t* *val*) {  printf("%s val is " PRI64d "\n", \_\_func\_\_, val); /\* Được cố định với chuỗi định dạng \*/  }  int main(void) {  /\* Khai báo các số nguyên với các độ rộng khác nhau \*/  *uint8\_t* u8 = 127;  *uint8\_t* s64 = INT64\_MAX;  /\* Đối số số nguyên được mở rộng khi tham số hàm có độ rộng lớn hơn \*/  param\_u8(u8); /\* param\_u8 val is 127 \*/  param\_u16(u8); /\* param\_u16 val is 127 \*/  param\_u32(u8); /\* param\_u32 val is 127 \*/  param\_u64(u8); /\* param\_u64 val is 127 \*/  param\_s8(u8); /\* param\_s8 val is 127 \*/  param\_s16(u8); /\* param\_s16 val is 127 \*/  param\_s32(u8); /\* param\_s32 val is 127 \*/  param\_s64(u8); /\* param\_s64 val is 127 \*/  /\* Đối số số nguyên bị cắt khi tham số hàm có độ rộng nhỏ hơn \*/  param\_u8(s64); /\* param\_u8 val is 255 \*/  param\_u16(s64); /\* param\_u16 val is 65535 \*/  param\_u32(s64); /\* param\_u32 val is 4294967295 \*/  param\_u64(s64); /\* param\_u64 val is 9223372036854775807 \*/  param\_s8(s64); /\* param\_s8 val is được xác định bởi cài đặt \*/  param\_s16(s64); /\* param\_s16 val is được xác định bởi cài đặt \*/  param\_s32(s64); /\* param\_s32 val is được xác định bởi cài đặt \*/  param\_s64(s64); /\* param\_s64 val is 9223372036854775807 \*/  return 0;  } |

## CHUYỂN ĐỔI CON TRỎ TRONG LỜI GỌI HÀM

Chuyển đổi con trỏ thành void\* là ngầm định (implicit), nhưng bất kỳ chuyển đổi con trỏ khác đều phải rõ ràng (explicit). Trong lập trình C, compiler cho phép thực hiện chuyển đổi rõ ràng từ bất kỳ con trỏ dữ liệu nào sang bất kỳ con trỏ dữ liệu khác, nhưng truy cập vào một đối tượng thông qua một con trỏ có kiểu sai là không hợp lệ và dẫn đến hành vi không xác định (undefined behavior). Điều duy nhất mà những trường hợp này được phép là khi các kiểu là tương thích hoặc khi con trỏ mà bạn sử dụng để xem đối tượng là một kiểu ký tự.

|  |
| --- |
| #include <stdio.h>  void func\_voidp(void\* *voidp*) {  printf("%s Address of ptr is %p\n", \_\_func\_\_, *voidp*);  }  struct *struct\_a* {  int a;  int b;  } data\_a;  struct *struct\_b* {  int a;  int b;  } data\_b;  void func\_struct\_b(struct *struct\_b*\* *bp*) {  printf("%s Address of ptr is %p\n", \_\_func\_\_, (void\*) *bp*);  }  int main(void) {  /\* Chuyển đổi con trỏ thành void\* được phép ngầm định \*/  func\_voidp(&data\_a);  /\*  \* Chuyển đổi con trỏ thành các kiểu khác cần được thực hiện rõ ràng  \*  \* Chú ý rằng ở đây mặc dù cả hai có định nghĩa giống nhau,  \* các kiểu không tương thích, và cuộc gọi này là không hợp lệ và dẫn đến hành vi không xác định khi chạy.  \*/  func\_struct\_b((struct *struct\_b*\*)&data\_a);  /\* Kết quả khi thực hiện chương trình có thể như sau: \*/  /\* func\_charp Address of ptr is 0x601030 \*/  /\* func\_voidp Address of ptr is 0x601030 \*/  /\* func\_struct\_b Address of ptr is 0x601030 \*/  return 0;  } |

# TYPE QUALIFIERS

Trong ngôn ngữ lập trình C, các type qualifier là các từ khóa (keyword) được sử dụng để sửa đổi các thuộc tính của các biến (variable). Sử dụng các type qualifier chúng ta có thể thay đổi các thuộc tính của các biến. Ngôn ngữ lập trình c cung cấp hai type qualifier là const và volatile. Xem thêm const type qualifier và volatile type qualifier để hiểu rõ thuật ngữ.

## VOLATILE VARIABLES

Từ khóa **"volatile"** thông báo cho trình biên dịch rằng giá trị của biến có thể thay đổi bất cứ lúc nào do các điều kiện bên ngoài, không chỉ do luồng điều khiển của chương trình.

Trình biên dịch sẽ không tối ưu hóa bất cứ điều gì liên quan đến biến volatile.

|  |
| --- |
| volatile int foo; /\* Cách khai báo biến volatile khác nhau \*/  int volatile foo;  volatile uint8\_t \* pReg; /\* Con trỏ tới biến volatile \*/  uint8\_t volatile \* pReg; |

Có hai lý do chính để sử dụng biến volatile:

* Giao tiếp với phần cứng có các thanh ghi I/O được ánh xạ vào bộ nhớ.
* Khi sử dụng các biến được sửa đổi bên ngoài luồng kiểm soát chương trình (ví dụ, trong một hàm xử lý ngắt).

Hãy xem ví dụ này:

|  |
| --- |
| int quit = false;  void main()  {  ...  while (!quit) {  // Do something that does not modify the quit variable  }  ...  }  void interrupt\_handler(void)  {  quit = true;  } |

Trình biên dịch có thể nhận thấy vòng lặp while không sửa đổi biến quit và chuyển vòng lặp thành vòng lặp vô hạn while (true). Ngay cả khi biến quit được đặt trong xử lý tín hiệu cho SIGINT và SIGTERM, trình biên dịch không biết điều đó.

Khai báo quit như volatile sẽ thông báo cho trình biên dịch không tối ưu hóa vòng lặp và vấn đề sẽ được giải quyết.

Vấn đề tương tự xảy ra khi truy cập phần cứng, như chúng ta thấy trong ví dụ này:

|  |
| --- |
| uint8\_t \* pReg = (uint8\_t \*) 0x1717;  // Chờ cho đến khi thanh ghi trở thành khác không  while (\*pReg == 0) { } // Làm một việc khác |

Hành vi của bộ tối ưu hóa là đọc giá trị biến một lần, không cần phải đọc lại, vì giá trị sẽ luôn luôn giống nhau. Vì vậy, chúng ta có vòng lặp vô hạn. Để buộc trình biên dịch thực hiện như chúng ta mong muốn, chúng ta sửa đổi khai báo thành:

|  |
| --- |
| uint8\_t volatile \* pReg = (uint8\_t volatile \*) 0x1717; |

## BIẾN KHÔNG THỂ SỬA ĐỔI (CONST)

|  |
| --- |
| const int a = 0; /\* Biến này là "không thể sửa đổi", trình biên dịch  sẽ báo lỗi khi biến này bị thay đổi \*/  int b = 0; /\* Biến này có thể sửa đổi \*/  b += 10; /\* Thay đổi giá trị của 'b' \*/  a += 10; /\* Gây ra lỗi từ trình biên dịch \*/ |

Tính chất "const" chỉ đơn giản là chúng ta không có quyền thay đổi dữ liệu. Điều này không có nghĩa rằng giá trị không thay đổi trong tình huống chúng ta không kiểm soát.

|  |
| --- |
| \_Bool doIt(double const\* *a*) {  double rememberA = \**a*;  // Làm một công việc dài và phức tạp gọi các hàm khác  return rememberA == \**a*;  } |

Trong quá trình thực thi của các cuộc gọi hàm khác, \*a có thể đã thay đổi, do đó hàm này có thể trả về cả **false** hoặc **true.**

**Cảnh báo:**

Biến có tính chất "const" vẫn có thể thay đổi bằng cách sử dụng con trỏ:

|  |
| --- |
| const int a = 0;  int \*a\_ptr = (int\*)&a; /\* Việc chuyển đổi này phải được thực hiện rõ ràng bằng cách ép kiểu \*/  \*a\_ptr += 10; /\* Điều này dẫn đến hành vi không xác định \*/  printf("a = %d\n", a); /\* Có thể in ra: "a = 10" \*/ |

Tuy nhiên, làm như vậy là một lỗi dẫn đến hành vi không xác định. Khó khăn ở đây là điều này có thể hoạt động đúng như mong đợi trong các ví dụ đơn giản như trên, nhưng lại gây sai lầm khi mã nguồn trở nên phức tạp hơn.

# TYPEDEF

Cơ chế typedef cho phép tạo ra các tên bí danh cho các kiểu khác. Nó không tạo ra các kiểu mới. Người ta thường sử dụng typedef để cải thiện tính di động của mã nguồn, đặt tên bí danh cho kiểu cấu trúc hoặc kiểu liên hợp, hoặc tạo bí danh cho kiểu hàm (hoặc con trỏ hàm).

Trong tiêu chuẩn C, typedef được phân loại như một 'lớp lưu trữ'(storage class) cho tính tiện dụng; nó xuất hiện cú pháp nơi các lớp lưu trữ như **static** hoặc **extern** có thể xuất hiện.

## TYPEDEF CHO STRUCTURES VÀ UNIONS

Bạn có thể đặt tên bí danh cho một cấu trúc:

|  |
| --- |
| typedef struct *Person* {  char name[32];  int age;  } *Person*;  *Person* person; |

So với cách truyền thống khi khai báo cấu trúc, người lập trình không cần phải sử dụng từ khóa "struct" mỗi khi họ khai báo một thể hiện của cấu trúc đó.

Lưu ý rằng tên "Person" (không phải struct Person) không được định nghĩa cho đến dấu chấm phẩy cuối cùng. Do đó, đối với các danh sách liên kết và cấu trúc cây mà cần chứa một con trỏ đến cùng kiểu cấu trúc, bạn phải sử dụng một trong hai cách:

|  |
| --- |
| typedef struct *Person* {  char name[32];  int age;  struct *Person* \*next;  } *Person*; |

Hoặc

|  |
| --- |
| typedef struct *Person* *Person*;  struct *Person* {  char name[32];  int age;  *Person* \*next;  }; |

Việc sử dụng typedef cho kiểu union cũng tương tự.

|  |
| --- |
| typedef union *Float* *Float*;  union *Float*  {  float f;  char b[sizeof(float)];  }; |

Một cấu trúc tương tự như thế này có thể được sử dụng để phân tích các byte tạo nên một giá trị float.

## TYPEDEF CHO CON TRỎ HÀM

Chúng ta có thể sử dụng typedef để đơn giản hóa việc sử dụng con trỏ hàm. Hãy tưởng tượng chúng ta có một số hàm, tất cả đều có cùng chữ ký, chúng sử dụng đối số của mình để in ra một cái gì đó theo cách khác nhau:

|  |
| --- |
| #include<stdio.h>  void print\_to\_n(int *n*)  {  for (int i = 1; i <= *n*; ++i)  printf("%d\n", i);  }  void print\_n(int *n*)  {  printf("%d\n", *n*);  } |

Bây giờ chúng ta có thể sử dụng typedef để tạo ra một kiểu con trỏ hàm có tên gọi là "printer":

|  |
| --- |
| typedef void (\**printer\_t*)(int); |

Điều này tạo ra một kiểu, được đặt tên là "printer\_t" cho một con trỏ trỏ đến một hàm có một đối số kiểu int và không trả về gì cả, phù hợp với chữ ký của các hàm chúng ta đã nêu trên. Để sử dụng nó, chúng ta tạo một biến của kiểu vừa tạo và gán cho nó một con trỏ trỏ tới một trong các hàm cần thiết:

|  |
| --- |
| *printer\_t* p = &print\_to\_n; |

Hoặc

|  |
| --- |
| void (\*p)(int) = &print\_to\_n; // Điều này cần thiết nếu không có kiểu |

Sau đó, để gọi hàm mà con trỏ hàm trỏ đến:

|  |
| --- |
| p(5); // In ra 1 2 3 4 5 trên các dòng riêng biệt  (\*p)(5); // Tương tự |

Do đó, typedef cho phép cú pháp đơn giản hơn khi làm việc với con trỏ hàm. Điều này trở nên rõ ràng hơn khi con trỏ hàm được sử dụng trong các tình huống phức tạp hơn, như là đối số cho các hàm.

Nếu bạn sử dụng một hàm nhận con trỏ hàm làm tham số mà không có kiểu con trỏ hàm được định nghĩa, định nghĩa hàm sẽ là:

|  |
| --- |
| void foo (void (\**printer*)(int), int *y*){  // mã  *printer*(*y*);  // mã  } |

Tuy nhiên, với typedef, nó trở thành:

|  |
| --- |
| void foo (*printer\_t* *printer*, int *y*){  // mã  *printer*(*y*);  // mã  } |

Tương tự, các hàm cũng có thể trả về con trỏ hàm và một lần nữa, việc sử dụng typedef có thể làm cú pháp đơn giản hơn khi làm như vậy.

Một ví dụ cổ điển là hàm signal trong <signal.h>. Khai báo của nó (từ tiêu chuẩn C) như sau:

|  |
| --- |
| void (\*signal(int *sig*, void (\**func*)(int)))(int); |

Đó là một hàm nhận hai đối số - một int và một con trỏ trỏ đến một hàm nhận một int làm đối số và không trả về gì cả - và trả về một con trỏ trỏ đến hàm giống như đối số thứ hai.

Nếu chúng ta định nghĩa một kiểu SigCatcher là một bí danh cho kiểu con trỏ trỏ đến hàm:

|  |
| --- |
| typedef void (\**SigCatcher*)(int); |

Sau đó, chúng ta có thể khai báo hàm signal() bằng cách sử dụng:

|  |
| --- |
| *SigCatcher* signal(int *sig*, *SigCatcher* *func*); |

Nhìn chung, điều này dễ hiểu hơn (ngay cả khi tiêu chuẩn C không định nghĩa một kiểu để thực hiện công việc này). Hàm signal nhận hai đối số, một int và một SigCatcher, và trả về một SigCatcher - trong đó SigCatcher là một con trỏ trỏ đến một hàm nhận đối số int và không trả về gì cả.

Mặc dù việc sử dụng tên typedef cho kiểu con trỏ hàm làm cho cuộc sống dễ dàng hơn, nhưng cũng có thể gây nhầm lẫn cho những người sau này sẽ duy trì mã nguồn của bạn, vì vậy hãy sử dụng cẩn thận và tài liệu đầy đủ. Xem thêm về con trỏ hàm.

## CÁC ỨNG DỤNG ĐƠN GIẢN CỦA TYPEDEF

### Để đặt tên ngắn gọn cho một kiểu dữ liệu

Thay vì viết:

|  |
| --- |
| long long int foo;  struct *mystructure* object; |

Chúng ta có thể sử dụng:

|  |
| --- |
| /\* khai báo một lần \*/  typedef long long *ll*;  typedef struct *mystructure* *mystruct*;  /\* sử dụng khi cần \*/  *ll* foo;  *mystruct* object; |

Điều này giảm số lượng gõ phím cần thiết nếu kiểu được sử dụng nhiều lần trong chương trình.

### Cải thiện tính di động

Các thuộc tính của các kiểu dữ liệu thay đổi trên các kiến trúc khác nhau. Ví dụ, một int có thể là một kiểu 2 byte trong một cài đặt và là một kiểu 4 byte trong một cài đặt khác. Giả sử một chương trình cần sử dụng một kiểu 4 byte để chạy đúng.

Trong một cài đặt, hãy giả sử kích thước của int là 2 byte và của long là 4 byte. Trong một cài đặt khác, hãy giả sử kích thước của int là 4 byte và của long là 8 byte. Nếu chương trình được viết bằng cách sử dụng cài đặt thứ hai,

|  |
| --- |
| /\* chương trình mong đợi một số nguyên 4 byte \*/  int foo; /\* cần giữ 4 byte để hoạt động \*/  /\* một số mã liên quan đến nhiều số nguyên hơn \*/ |

Để chương trình chạy trong cài đặt đầu tiên, tất cả các khai báo **int** sẽ phải được thay đổi thành **long**.

|  |
| --- |
| /\* chương trình cần sử dụng long bây giờ \*/  long foo; /\* cần giữ 4 byte để hoạt động \*/  /\* một số mã liên quan đến nhiều long hơn - phải thay đổi nhiều \*/ |

Để tránh điều này, chúng ta có thể sử dụng **typedef**

|  |
| --- |
| /\* chương trình mong đợi một số nguyên 4 byte \*/  typedef int *myint*; /\* cần khai báo một lần - chỉ cần thay đổi một dòng nếu cần \*/  *myint* foo; /\* cần giữ 4 byte để hoạt động \*/  /\* một số mã liên quan đến nhiều myint hơn \*/ |

Sau đó, chỉ cần thay đổi câu lệnh typedef mỗi lần, thay vì xem xét toàn bộ chương trình.

Phiên bản ≥ C99

Tiêu chuẩn <stdint.h> và tiêu chuẩn liên quan <inttypes.h> xác định tên kiểu tiêu chuẩn (sử dụng typedef) cho các số nguyên có kích thước khác nhau, và những tên này thường là lựa chọn tốt trong mã hiện đại cần các số nguyên có kích thước cố định. Ví dụ, uint8\_t là kiểu số nguyên không dấu 8-bit; int64\_t là kiểu số nguyên có dấu 64-bit. Kiểu uintptr\_t là kiểu số nguyên không dấu đủ lớn để giữ bất kỳ con trỏ nào tới đối tượng. Những kiểu này lý thuyết là tùy chọn - nhưng hiếm khi chúng không khả dụng. Có các biến thể như uint\_least16\_t (kiểu số nguyên không dấu nhỏ nhất có ít nhất 16 bit) và int\_fast32\_t (kiểu số nguyên có dấu nhanh nhất có ít nhất 32 bit). Ngoài ra, intmax\_t và uintmax\_t là các kiểu số nguyên lớn nhất được hỗ trợ bởi triển khai. Những kiểu này là bắt buộc.

### Để chỉ định cách sử dụng hoặc cải thiện khả năng đọc

Nếu một tập dữ liệu có mục đích cụ thể, bạn có thể sử dụng typedef để đặt tên có ý nghĩa cho nó. Hơn nữa, nếu thuộc tính của dữ liệu thay đổi sao cho kiểu cơ bản phải thay đổi, chỉ cần thay đổi câu lệnh typedef, thay vì xem xét toàn bộ chương trình.

# LỚP LƯU TRỮ (STORAGE CLASS)

Lớp lưu trữ được sử dụng để đặt phạm vi của một biến hoặc hàm. Bằng cách biết lớp lưu trữ của một biến, chúng ta có thể xác định thời gian tồn tại của biến đó trong thời gian chạy của chương trình.

Có các lớp lưu trữ sau:

1. Auto
2. Register
3. Static
4. Typedef

## Auto

Lớp lưu trữ này chỉ ra rằng một định danh có thời gian lưu trữ tự động. Điều này có nghĩa rằng sau khi phạm vi hoạt động mà định danh được định nghĩa kết thúc, đối tượng được đại diện bởi định danh không còn hợp lệ nữa.

Vì tất cả các đối tượng, không nằm trong phạm vi toàn cục hoặc được khai báo tĩnh, đều có thời gian lưu trữ tự động theo mặc định khi được định nghĩa, từ khóa này chủ yếu có ý nghĩa lịch sử và không nên được sử dụng:

|  |
| --- |
| int foo(void) {  /\* Một số nguyên có thời gian lưu trữ tự động. \*/  auto int i = 3;  /\* Tương tự \*/  int j = 5;  return 0;  } /\* Các giá trị của i và j không còn có thể sử dụng được nữa. \*/ |

Tóm lại, lớp lưu trữ "auto" không còn phổ biến trong mã nguồn hiện đại và thường không nên sử dụng. Các biến được khai báo trong phạm vi hàm sẽ mặc định có thời gian lưu trữ tự động.

## Register

Dùng để gợi ý cho trình biên dịch rằng việc truy cập vào một đối tượng nên nhanh nhất có thể. Việc trình biên dịch thực sự sử dụng gợi ý này là do triển khai cụ thể; nó có thể đơn giản là xem như tương đương với auto.

Thuộc tính duy nhất mà chắc chắn khác nhau cho tất cả các đối tượng được khai báo với từ khóa register là chúng không thể có địa chỉ của chúng tính toán. Do đó, register có thể là một công cụ tốt để đảm bảo một số tối ưu hóa cụ thể:

|  |
| --- |
| register size\_t size = 467; |

là một đối tượng không bao giờ có khả năng trùng tên vì không có mã nào có thể truyền địa chỉ của nó cho một hàm khác mà nó có thể bị thay đổi một cách không mong muốn.

Thuộc tính này cũng ngụ ý rằng một mảng

|  |
| --- |
| register int array[5]; |

không thể phân rã thành con trỏ tới phần tử đầu tiên của nó (tức là mảng không thể biến thành &array[0]). Điều này có nghĩa là các phần tử của một mảng như vậy không thể truy cập và chính mảng cũng không thể được truyền vào một hàm.

Trong thực tế, việc sử dụng hợp lệ duy nhất của một mảng được khai báo với lớp lưu trữ register là toán tử sizeof; bất kỳ toán tử nào khác đều yêu cầu địa chỉ của phần tử đầu tiên của mảng. Vì lý do đó, mảng thường không nên được khai báo với từ khóa register vì nó khiến chúng trở nên vô dụng cho bất kỳ điều gì ngoài việc tính toán kích thước của toàn bộ mảng, điều này có thể thực hiện dễ dàng mà không cần từ khóa register.

Lớp lưu trữ register thích hợp hơn cho các biến được định nghĩa trong một khối và được truy cập với tần suất cao. Ví dụ,

|  |
| --- |
| /\* in ra tổng của 5 số nguyên đầu tiên \*/  /\* mã giả định là một phần của thân hàm \*/  {  register int k, sum;  for(k = 1, sum = 0; k < 6; sum += k, k++);  printf("\t%d\n",sum);  } |

Phiên bản ≥ C11

Toán tử \_Alignof cũng được phép được sử dụng với mảng có lớp lưu trữ register.

## STATIC

Lớp lưu trữ static có các mục đích khác nhau, phụ thuộc vào vị trí khai báo trong tệp:

1. Để giới hạn định danh chỉ cho đơn vị dịch thuật đó (phạm vi=tệp).

|  |
| --- |
| /\* Không có đơn vị dịch thuật nào khác có thể sử dụng biến này. \*/  static int i;  /\* Tương tự; static được gắn vào kiểu hàm của f, không phải kiểu trả về int. \*/  static int f(int *n*); |

1. Để lưu trữ dữ liệu để sử dụng cho lần gọi hàm kế tiếp (phạm vi=khối):

|  |
| --- |
| void foo()  {  static int a = 0; /\* có thời gian lưu trữ static và tuổi thọ của nó là  \* toàn bộ thực thi của chương trình; được khởi tạo thành 0 trong  \* lần gọi hàm đầu tiên \*/  int b = 0; /\* b có phạm vi khối và thời gian lưu trữ tự động và  \* chỉ "tồn tại" trong hàm \*/  a += 10;  b += 10;  printf("static int a = %d, int b = %d\n", a, b);  }  int main(void)  {  int i;  for (i = 0; i < 5; i++)  {  foo();  }  return 0;  } |

Mã này in ra

|  |
| --- |
| static int a = 10, int b = 10  static int a = 20, int b = 10  static int a = 30, int b = 10  static int a = 40, int b = 10  static int a = 50, int b = 10 |

Biến static giữ lại giá trị của nó ngay cả khi được gọi từ nhiều luồng khác nhau.

1. Được sử dụng trong tham số hàm để chỉ ra mảng dự kiến ​​có số lượng phần tử tối thiểu là hằng số và một tham số không phải con trỏ null:

|  |
| --- |
| /\* a dự kiến ​​có ít nhất 512 phần tử. \*/  void printInts(int *a*[static 512])  {  size\_t i;  for (i = 0; i < 512; ++i)  printf("%d\n", *a*[i]);  } |

Số lượng phần tử yêu cầu (hoặc thậm chí một con trỏ không phải null) không nhất thiết phải được kiểm tra bởi trình biên dịch và trình biên dịch không yêu cầu thông báo cho bạn bằng bất kỳ cách nào nếu bạn không có đủ phần tử. Nếu một người lập trình truyền ít hơn 512 phần tử hoặc một con trỏ null, kết quả sẽ là hành vi không xác định. Vì không thể thực thi điều này, cần phải sử dụng cẩn thận hơn khi truyền giá trị cho tham số đó tới một hàm như vậy.

## TYPEDEF

Định nghĩa một kiểu mới dựa trên một kiểu hiện có. Cú pháp của nó phản ánh cú pháp của một khai báo biến.

|  |
| --- |
| /\* Byte có thể được sử dụng bất cứ nơi nào cần `unsigned char` \*/  typedef unsigned char *Byte*;  /\* Integer là kiểu được sử dụng để khai báo một mảng gồm một int \*/  typedef int *Integer*[1];  /\* NodeRef là kiểu được sử dụng cho con trỏ tới kiểu cấu trúc có thẻ "node" \*/  typedef struct *node* \**NodeRef*;  /\* SigHandler là kiểu con trỏ hàm được truyền vào hàm signal. \*/  typedef void (\**SigHandler*)(int); |

Mặc dù không phải lớp lưu trữ kỹ thuật, trình biên dịch sẽ xem nó như là một lớp lưu trữ vì không có lớp lưu trữ khác được phép nếu sử dụng từ khóa typedef.

Các typedef rất quan trọng và không nên được thay thế bằng #define macro.

|  |
| --- |
| typedef int *newType*;  *newType* \*ptr; // ptr là con trỏ đến biến có kiểu 'newType' tương đương int |

Tuy nhiên:

|  |
| --- |
| #define int *newType*  *newType* \*ptr; // Mặc dù macro là thay thế chính xác cho các từ, điều này không dẫn đến  // một con trỏ đến biến có kiểu 'newType' tương đương int |

## EXTERN

Sử dụng để khai báo một đối tượng hoặc hàm đã được định nghĩa ở nơi khác (và có sự liên kết ngoại). Nó thường được sử dụng để khai báo một đối tượng hoặc hàm sẽ được sử dụng trong một module mà không phải là module trong đó đối tượng hoặc hàm tương ứng được định nghĩa:

|  |
| --- |
| /\* file1.c \*/  int foo = 2; /\* Có sự liên kết ngoại vì nó được khai báo ở phạm vi tệp. \*/  /\* file2.c \*/  #include <stdio.h>  int main(void)  {  /\* Từ khóa `extern` ám chỉ đến định nghĩa bên ngoài của `foo`. \*/  extern int foo;  printf("%d\n", foo);  return 0;  } |

Phiên bản ≥ C99

Mọi thứ trở nên thú vị hơn một chút với việc giới thiệu từ khóa **inline** trong C99:

|  |
| --- |
| /\* Nên đặt trong một tệp tiêu đề để tất cả người dùng thấy định nghĩa \*/  /\* Gợi ý cho trình biên dịch rằng hàm `bar` có thể được nội tuyến \*/  /\* và ẩn biểu tượng ngoại cung, trừ khi nêu rõ khác. \*/  inline void bar(int *drink*)  {  printf("Bạn đã đặt món uống số %d\n", *drink*);  }  /\* Sẽ có trong chỉ một tệp .c.  Tạo một định nghĩa hàm ngoại cung của `bar` để sử dụng bởi các tệp khác.  Trình biên dịch được phép chọn giữa phiên bản nội tuyến và định nghĩa ngoại cung  khi gọi `bar`. Nếu không có dòng này, `bar` sẽ chỉ là một hàm nội tuyến,  và các tệp khác sẽ không thể gọi nó. \*/  extern void bar(int); |

## \_THREAD\_LOCAL

Phiên bản ≥ C11

Đây là một mẫu lưu trữ mới được giới thiệu trong C11 cùng với đa luồng. Điều này không có sẵn trong các tiêu chuẩn C trước đó.

Được sử dụng để chỉ định thời gian lưu trữ của luồng. Một biến được khai báo với mẫu lưu trữ \_Thread\_local cho biết đối tượng chỉ cục bộ đối với luồng đó và tuổi thọ của nó là toàn bộ thực thi của luồng mà nó được tạo ra. Nó cũng có thể xuất hiện cùng với static hoặc extern.

|  |
| --- |
| #include <threads.h>  #include <stdio.h>  #define SIZE 5  int thread\_func(void \**id*)  {  /\* biến cục bộ của luồng i. \*/  static \_Thread\_local int i;  /\* In ra ID được truyền từ main() và địa chỉ của i.  \* Chạy chương trình này sẽ in ra các địa chỉ khác nhau cho i, cho thấy  \* rằng chúng là các đối tượng riêng biệt. \*/  printf("Từ luồng:[%d], Địa chỉ của i (cục bộ luồng): %p\n", \*(int\*)*id*, (void\*)&i);  return 0;  }  int main(void)  {  *thrd\_t* id[SIZE];  int arr[SIZE] = {1, 2, 3, 4, 5};  /\* tạo 5 luồng. \*/  for(int i = 0; i < SIZE; i++) {  thrd\_create(&id[i], thread\_func, &arr[i]);  }  /\* đợi các luồng hoàn thành. \*/  for(int i = 0; i < SIZE; i++) {  thrd\_join(id[i], NULL);  }  } |

.

# KHAI BÁO

## GỌI MỘT HÀM TỪ MỘT TỆP C KHÁC

**foo.h**

|  |
| --- |
| #ifndef FOO\_DOT\_H /\* Đây là "include guard" \*/  #define FOO\_DOT\_H /\* ngăn tệp bị bao gồm hai lần. \*/  /\* Bao gồm một tệp tiêu đề hai lần gây ra tất cả loại \*/  /\* các vấn đề thú vị. \*/  /\*  \* Đây là một khai báo hàm.  \* Nó thông báo cho trình biên dịch rằng hàm tồn tại ở một nơi nào đó.  \*/  void foo(int *id*, char \**name*);  #endif /\* FOO\_DOT\_H \*/ |

**foo.c**

|  |
| --- |
| #include "foo.h" /\* Luôn bao gồm tệp tiêu đề khai báo điều gì đó  \* trong tệp C định nghĩa nó. Điều này đảm bảo rằng  \* khai báo và định nghĩa luôn được đồng bộ. Đặt tiêu đề này  \* ở đầu foo.c để đảm bảo tiêu đề tự chứa các thông tin.  \*/  #include <stdio.h>  /\*\*  \* Đây là định nghĩa hàm.  \* Đó là phần thực tế của hàm đã được khai báo ở nơi khác.  \*/  void foo(int *id*, char \**name*)  {  fprintf(stderr, "foo(%d, \"%s\");\n", *id*, *name*);  /\* Điều này sẽ in ra cách foo được gọi ra standard error.  \* ví dụ, foo(42, "Hi!") sẽ in ra `foo(42, "Hi!")`  \*/  } |

**main.c**

|  |
| --- |
| #include "foo.h"  int main(void)  {  foo(42, "bar");  return 0;  } |

**Compile and Link**

Trước hết, chúng ta biên dịch cả **foo.c** và **main.c** thành các tệp đối tượng. Ở đây, chúng tôi sử dụng trình biên dịch gcc, trình biên dịch của bạn có thể có tên khác và cần các tùy chọn khác.

|  |
| --- |
| $ gcc -Wall -c foo.c  $ gcc -Wall -c main.c |

Bây giờ chúng ta liên kết chúng lại với nhau để tạo ra chương trình thực thi cuối cùng:

|  |
| --- |
| $ gcc -o testprogram foo.o main.o |

## SỬ DỤNG BIẾN TOÀN CỤC

Việc sử dụng biến toàn cục thường không được khuyến khích. Nó làm cho chương trình của bạn khó hiểu hơn và khó khăn hơn trong việc sửa lỗi. Tuy nhiên, đôi khi việc sử dụng biến toàn cục là chấp nhận được.

**global.h**

|  |
| --- |
| #ifndef GLOBAL\_DOT\_H /\* Đây là "include guard" \*/  #define GLOBAL\_DOT\_H  /\*\*  \* Điều này thông báo cho trình biên dịch rằng biến g\_myglobal tồn tại ở một nơi nào đó.  \* Mà không có "extern", điều này sẽ tạo ra một biến mới có tên là  \* g\_myglobal trong \_mọi tệp\_ mà bao gồm nó. Đừng bỏ lỡ điều này!  \*/  extern int g\_myglobal; /\* \_Khai báo\_ g\_myglobal, nghĩa là hứa rằng nó sẽ được \_định nghĩa\_ bởi  \* một số module nào đó. \*/    #endif /\* GLOBAL\_DOT\_H \*/ |

**global.c**

|  |
| --- |
| #include "global.h" /\* Luôn bao gồm tệp tiêu đề khai báo điều gì đó  \* trong tệp C định nghĩa nó. Điều này đảm bảo rằng  \* khai báo và định nghĩa luôn được đồng bộ.  \*/    int g\_myglobal; /\* \_Định nghĩa\_ my\_global. Vì nó sống ở phạm vi toàn cục nó được khởi tạo thành 0  \* khi chương trình khởi động. \*/ |

**main.c**

|  |
| --- |
| #include "global.h"  int main(void)  {  g\_myglobal = 42;  return 0;  } |

Xem thêm: Làm thế nào để sử dụng **extern** để chia sẻ biến giữa các tệp nguồn?

## GIỚI THIỆU

Ví dụ về các khai báo là

|  |
| --- |
| int a; /\* khai báo một cái tên đơn thuộc kiểu int \*/ |

Khai báo trên khai báo một cái tên đơn được gọi là "a" mà tham chiếu đến một đối tượng có kiểu int.

|  |
| --- |
| int a1, b1; /\* khai báo 2 cái tên thuộc kiểu int \*/ |

Khai báo thứ hai khai báo 2 cái tên là "a1" và "b1" mà tham chiếu đến các đối tượng khác nhau, nhưng cùng có kiểu int.

Cách thức hoạt động của nó cơ bản là như sau - trước hết, bạn đặt một số kiểu dữ liệu, sau đó bạn viết một biểu thức duy nhất hoặc nhiều biểu thức cách nhau bằng dấu phẩy (,) (các biểu thức này sẽ không được đánh giá tại thời điểm này - và trong ngữ cảnh này, các biểu thức này nên được gọi là khai báo (declarators). Khi viết các biểu thức như vậy, bạn chỉ được áp dụng các toán tử **indirection ()**, **function call ()** hoặc **subscript (hoặc array indexing - )** lên một số mã định danh (bạn cũng có thể không sử dụng bất kỳ toán tử nào cả). Mã định danh được sử dụng không cần phải hiển thị trong phạm vi hiện tại. Mộtsố ví dụ:

|  |
| --- |
| int /\* 1 \*/ (\*z) /\* 2 \*/ , /\* 3 \*/ \*x , /\* 4 \*/ \*\*c /\* 6 \*/ ; |

|  |  |
| --- | --- |
| # | Sự miêu tả |
| 1 | Tên của kiểu số nguyên. |
| 2 | Biểu thức chưa được đánh giá áp dụng gián tiếp cho một số định danh .z |
| 3 | Chúng tôi có dấu phẩy chỉ ra rằng một biểu thức nữa sẽ theo sau trong cùng một khai báo. |
| 4 | Biểu thức chưa được đánh giá áp dụng gián tiếp cho một số mã định danh khác .x |
| 5 | Biểu thức chưa được đánh giá áp dụng gián tiếp cho giá trị của biểu thức .(\*c) |
| 6 | Kết thúc của khai báo. |

Chú ý rằng không có một trong những cái tên trên được hiển thị trước khi khai báo này, vì vậy các biểu thức được sử dụng không hợp lệ trước nó.

Sau mỗi biểu thức như vậy, cái tên được sử dụng trong nó sẽ được giới thiệu vào phạm vi hiện tại. (Nếu cái tên có liên kết được gán cho nó, nó cũng có thể được khai báo lại với cùng một kiểu liên kết để cả hai cái tên đều tham chiếu đến cùng một đối tượng hoặc hàm)

Ngoài ra, toán tử bằng (=) có thể được sử dụng để khởi tạo. Nếu một biểu thức chưa được đánh giá (declarator) được theo sau bởi dấu bằng trong khai báo - chúng ta nói rằng cái tên đang được giới thiệu cũng đang được khởi tạo. Sau dấu bằng chúng ta có thể đặt một lần nữa một số biểu thức, nhưng lần này nó sẽ được đánh giá và giá trị của nó sẽ được sử dụng làm giá trị ban đầu cho đối tượng được khai báo.

Ví dụ:

|  |
| --- |
| int l = 90; /\* tương đương với: \*/  int l; l = 90; /\* nếu khai báo của l ở trong phạm vi khối \*/  int c = 2, b[c]; /\* ok, tương đương với: \*/  int c = 2; int b[c]; |

Trong phần mã của bạn sau này, bạn được phép viết biểu thức chính xác giống như phần khai báo của cái tên mới được giới thiệu, cho bạn một đối tượng có kiểu được xác định ở đầu khai báo, giả định rằng bạn đã gán giá trị hợp lệ cho tất cả các đối tượng được truy cập theo cách đó. Ví dụ:

|  |
| --- |
| void f()  {  int b2; /\* bạn nên có thể viết sau này trong mã của bạn b2  mà sẽ trực tiếp tham chiếu đến đối tượng số nguyên  mà b2 định danh \*/    b2 = 2; /\* gán giá trị cho b2 \*/    printf("%d", b2); /\*ok - nên in 2\*/  int \*b3; /\* bạn nên có thể viết sau này trong mã của bạn \*b3 \*/  b3 = &b2; /\* gán giá trị con trỏ hợp lệ cho b3 \*/  printf("%d", \*b3); /\* ok - nên in 2 \*/  int \*\*b4; /\* bạn nên có thể viết sau này trong mã của bạn \*\*b4 \*/  b4 = &b3;  printf("%d", \*\*b4); /\* ok - nên in 2 \*/  void (\*p)(); /\* bạn nên có thể viết sau này trong mã của bạn (\*p)() \*/  p = &f; /\* gán một giá trị con trỏ hợp lệ \*/  (\*p)(); /\* ok - gọi hàm f bằng cách lấy giá trị con trỏ bên trong p - p  và giải tham chiếu nó - \*p  kết quả là một hàm  sau đó được gọi - (\*p)() -  nó không phải là \*p() vì nếu không trước hết toán tử () được áp dụng vào p và sau đó đối tượng void kết quả được giải tham chiếu không phải là những gì chúng ta muốn ở đây \*/  } |

Khai báo của b3 xác định rằng bạn có thể tiềm ẩn sử dụng giá trị của b3 như một phương tiện để truy cập vào một số đối tượng số nguyên. Tất nhiên, để áp dụng con trỏ (\*) vào b3, bạn cũng cần có giá trị thích hợp được lưu trữ trong nó (xem thêm về con trỏ để biết thêm thông tin). Bạn cũng nên trước tiên lưu một số giá trị vào một đối tượng trước khi cố gắng truy xuất nó (bạn có thể xem thêm về vấn đề này tại đây). Chúng ta đã thực hiện tất cả các điều này trong các ví dụ trên.

|  |
| --- |
| int a3(); /\* bạn nên có thể gọi a3 \*/ |

Thành phần này cho trình biên dịch biết rằng bạn sẽ cố gắng gọi a3. Trong trường hợp này, a3 tham chiếu đến một hàm thay vì một đối tượng. Một khác biệt giữa đối tượng và hàm là các hàm luôn luôn có một số loại liên kết. Ví dụ:

|  |
| --- |
| void f1()  {  {  int f2(); /\* 1 tham chiếu đến một số hàm f2 \*/  }    {  int f2(); /\* tham chiếu đến cùng một hàm f2 như (1) \*/  }  } |

Trong ví dụ trên, 2 khai báo tham chiếu đến cùng một hàm f2, trong khi nếu chúng đang khai báo đối tượng thì trong ngữ cảnh này (có 2 phạm vi khối khác nhau), chúng sẽ được coi là 2 đối tượng riêng biệt khác nhau.

|  |
| --- |
| int (\*a3)(); /\* bạn nên có thể áp dụng con trỏ cho `a3` và sau đó gọi nó \*/ |

Bây giờ có vẻ như nó đang trở nên phức tạp, nhưng nếu bạn biết về thứ tự ưu tiên của toán tử, bạn sẽ không gặp vấn đề gì khi đọc các khai báo trên. Các dấu ngoặc đơn cần thiết vì toán tử \* có mức ưu tiên thấp hơn toán tử (). Trong trường hợp sử dụng toán tử chỉ số, biểu thức kết quả không thể được xác định thực sự sau phần khai báo vì chỉ số được sử dụng trong nó (giá trị trong [ và ]) sẽ luôn luôn cao hơn giá trị tối đa được phép cho đối tượng/hàm này.

|  |
| --- |
| int a4[5]; /\* ở đây a4 không nên được truy cập bằng chỉ số 5 sau này \*/ |

Nhưng nó nên được truy cập bằng tất cả các chỉ số khác nhau dưới 5. Ví dụ:

|  |
| --- |
| a4[0], a4[1]; a4[4]; |

a4[5] sẽ dẫn đến hành vi không xác định (undefined behavior).

Thông tin thêm về mảng có thể được tìm thấy ở đây.

|  |
| --- |
| int (\*a5)[5](); /\* here a4 could be applied indirection  indexed up to (but not including) 5  and called \* |

Thật không may cho chúng tôi, mặc dù về mặt cú pháp có thể đúng, việc tuyên bố bị cấm theo tiêu chuẩn hiện tại.a5

## TYPEDEF

Khái niệm typedef là một loại khai báo có từ khóa **typedef** đứng trước và trước kiểu dữ liệu. Ví dụ:

|  |
| --- |
| typedef int (\*(\**t0*)())[5]; |

(Bạn cũng có thể đặt từ khóa **typedef** sau kiểu dữ liệu - như **int typedef (\*(\*t0)())[5];** nhưng điều này không được khuyến khích)

Phần khai báo trên định nghĩa một định danh cho một tên kiểu dữ liệu **typedef**. Sau đó, bạn có thể sử dụng nó như sau:

|  |
| --- |
| *t0* pf; |

Điều này có tác dụng tương tự như việc viết:

|  |
| --- |
| int (\*(\*pf)())[5]; |

Như bạn có thể thấy, tên kiểu dữ liệu typedef "lưu giữ" khai báo như một kiểu để sử dụng sau đó trong các khai báo khác. Điều này giúp bạn tiết kiệm một số lần nhấn phím. Bên cạnh đó, việc sử dụng typedef để khai báo vẫn giữ tính chất khai báo, do đó bạn không bị giới hạn chỉ bởi ví dụ trên:

|  |
| --- |
| *t0* (\**pf1*); |

Tương đương với:

|  |
| --- |
| int (\*(\*\*pf1)())[5]; |

Như bạn có thể thấy, khai báo typedef cho phép bạn tạo ra các tên kiểu dữ liệu tùy chỉnh để sử dụng trong mã của bạn, làm cho mã dễ đọc hơn và dễ duy trì hơn, đặc biệt khi làm việc với các kiểu dữ liệu phức tạp.

## SỬ DỤNG HẰNG SỐ TOÀN CỤC

Các tệp tiêu đề có thể được sử dụng để khai báo các tài nguyên chỉ đọc toàn cục, chẳng hạn như bảng chuỗi.

Hãy khai báo chúng trong một tệp tiêu đề riêng biệt, sau đó bao gồm tệp đó vào bất kỳ tệp nào ("Đơn vị Dịch") muốn sử dụng chúng. Thường thì cùng một tiêu đề cũng được sử dụng để khai báo một enumeration liên quan để xác định tất cả các nguồn tài nguyên chuỗi:

**resources.h:**

|  |
| --- |
| #ifndef RESOURCES\_H  #define RESOURCES\_H  typedef enum {  RESOURCE\_UNDEFINED = -1,  RESOURCE\_UNKNOWN = 0,  RESOURCE\_OK,  RESOURCE\_CANCEL,  RESOURCE\_ABORT,  RESOURCE\_MAX  } *EnumResourceID*;  extern const char \* const resources[RESOURCE\_MAX];  #endif |

Để thực sự định nghĩa các nguồn tài nguyên, tạo một tệp .c liên quan, đó là một đơn vị dịch khác chứa các phiên bản thực tế của những gì đã được khai báo trong tệp tiêu đề (.h):

**resources.c:**

|  |
| --- |
| #include "resources.h"  const char \* const resources[RESOURCE\_MAX] = {  "<unknown>",  "OK",  "Cancel",  "Abort"  }; |

Một chương trình sử dụng này có thể như sau:

**main.c:**

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include "resources.h"  int main(void)  {  *EnumResourceID* resource\_id = RESOURCE\_UNDEFINED;  while ((++resource\_id) < RESOURCE\_MAX)  {  printf("resource ID: %d, resource: '%s'\n", resource\_id, resources[resource\_id]);  }  return EXIT\_SUCCESS; |

Biên dịch ba tệp trên bằng GCC và liên kết chúng để trở thành tệp chương trình main chẳng hạn như sau:

|  |
| --- |
| gcc -Wall -Wextra -pedantic -Wconversion -g main.c resources.c -o main |

Chạy chương trình đã tạo

|  |
| --- |
| $ ./main |

Và kết quả nhận được sẽ là:

|  |
| --- |
| resource ID: 0, resource: ''  resource ID: 1, resource: 'OK'  resource ID: 2, resource: 'Cancel'  resource ID: 3, resource: 'Abort' |

## SỬ DỤNG QUY TẮC PHẢI TRÁI HOẶC XOẮN ỐC ĐỂ GIẢI MÃ CÁC KHAI BÁO TRONG C

Quy tắc "phải-trái" là một quy tắc hoàn toàn thông thường để giải mã các khai báo C. Nó cũng có thể hữu ích trong việc tạo ra chúng.

Đọc các biểu tượng khi bạn gặp chúng trong khai báo...

|  |
| --- |
| \* như "con trỏ tới" - luôn ở phía trái  [] như "mảng của" - luôn ở phía phải  () như "hàm trả về" - luôn ở phía phải |

**Cách áp dụng quy tắc:**

**BƯỚC 1**

Tìm cái định danh. Đây là điểm khởi đầu của bạn. Sau đó, tự nói với bản thân, "cái định danh là." Bạn đã bắt đầu khai báo của mình.

**BƯỚC 2**

Nhìn vào các biểu tượng ở bên phải của cái định danh. Ví dụ, nếu bạn thấy () ở đó, thì bạn biết đó là khai báo cho một hàm. Vì vậy, bạn sẽ có "cái định danh là hàm trả về". Hoặc nếu bạn thấy [] ở đó, bạn sẽ nói "cái định danh là mảng của". Tiếp tục sang phải cho đến khi bạn hết biểu tượng HOẶC gặp dấu ngoặc phải ). (Nếu bạn gặp dấu ngoặc trái (, đó là phần đầu của biểu tượng (), ngay cả khi có gì đó ở giữa các dấu ngoặc. Thêm thông tin về điều này ở dưới.)

**BƯỚC 3**

Nhìn vào các biểu tượng bên trái của cái định danh. Nếu nó không phải là một trong những biểu tượng ở trên (ví dụ, "int"), chỉ cần nói nó. Nếu không, dịch nó sang tiếng Anh bằng bảng ở trên. Tiếp tục sang trái cho đến khi bạn hết biểu tượng HOẶC gặp dấu ngoặc trái (.

Bây giờ lặp lại các bước 2 và 3 cho đến khi bạn đã tạo ra khai báo của mình.

**Dưới đây là một số ví dụ:**

|  |
| --- |
| int \*p[]; |

Trước tiên, tìm cái định danh:

|  |
| --- |
| int \*p[];  ^ |

“là p”

Tiếp theo, di chuyển sang phải cho đến khi hết biểu tượng hoặc gặp dấu ngoặc phải.

|  |
| --- |
| int \*p[];  ^^ |

“p là mảng của”

Không thể di chuyển sang phải nữa (hết biểu tượng), vậy di chuyển sang trái và tìm:

|  |
| --- |
| int \*p[];  ^ |

"p là mảng của con trỏ tới"

Tiếp tục sang trái và tìm:

|  |
| --- |
| int \*p[];  ^^^ |

"p là mảng của con trỏ tới int".

(hoặc "p là một mảng trong đó mỗi phần tử thuộc kiểu con trỏ đến int")

**Hoặc một ví dụ khác:**

|  |
| --- |
| int \*(\*func())(); |

Tìm cái định danh

|  |
| --- |
| int \*(\*func())();  ^^^^ |

“là **func**”

Di chuyển sang phải

|  |
| --- |
| int \*(\*func())();  ^^ |

"**func** là hàm trả về"

Không thể di chuyển sang phải nữa vì có dấu ngoặc phải, vậy di chuyển sang trái.

|  |
| --- |
| int \*(\*func())();  ^ |

"**func** là hàm trả về con trỏ trỏ tới"

Không thể di chuyển sang trái nữa vì có dấu ngoặc trái, vậy tiếp tục sang phải.

|  |
| --- |
| int \*(\*func())();  ^^ |

"**func** là hàm trả về con trỏ tới hàm trả về"

Không thể di chuyển sang phải nữa vì chúng ta hết biểu tượng, vậy di chuyển sang trái.

|  |
| --- |
| int \*(\*func())();  ^ |

"**func** là hàm trả về con trỏ tới hàm trả về con trỏ tới"

Và cuối cùng, tiếp tục sang trái, vì không còn gì ở bên phải.

|  |
| --- |
| int \*(\*func())();  ^^^ |

"**func** là hàm trả về con trỏ tới hàm trả về con trỏ trỏ tới int".

Như bạn có thể thấy, quy tắc này có thể rất hữu ích. Bạn cũng có thể sử dụng nó để kiểm tra lý thuyết của mình trong khi bạn tạo các khai báo và để cho bạn một gợi ý về nơi đặt biểu tượng tiếp theo và xem liệu có cần dấu ngoặc đơn hay không.

Một số khai báo trông phức tạp hơn so với thực sự của chúng do kích thước mảng và danh sách đối số trong dạng nguyên mẫu. Nếu bạn thấy **[3]**, đó được đọc như **"mảng (kích thước 3)** của...". Nếu bạn thấy **(char \*, int)**, đó được đọc như **\*"hàm mong đợi (char ,int) và trả về...".**

**Dưới đây là một ví dụ thú vị**:

|  |
| --- |
| int (\*(\*fun\_one)(char \*,double))[9][20]; |

Tôi sẽ không đi qua từng bước để giải mã điều này.

"fun\_one là con trỏ tới hàm mong đợi (char ,double) và trả về con trỏ tới mảng (kích thước 9) của mảng (kích thước 20) của int."

Như bạn có thể thấy, nó không phức tạp nếu bạn loại bỏ kích thước mảng và danh sách đối số:

|  |
| --- |
| int (\*(\*fun\_one)())[][]; |

Bạn có thể giải mã nó như cách đó và sau đó thêm kích thước mảng và danh sách đối số sau.

**Một số lời kết:**

Hoàn toàn có thể tạo các khai báo không hợp lệ bằng cách sử dụng quy tắc này, vì vậy cần có một số kiến thức về điều gì hợp lệ trong C. Ví dụ, nếu cái trên là:

|  |
| --- |
| int \*((\*fun\_one)())[][]; |

nó sẽ đọc "fun\_one là con trỏ tới hàm trả về mảng của mảng của con trỏ tới int". Vì một hàm không thể trả về một mảng, mà chỉ có thể trả về một con trỏ tới mảng, khai báo đó là không hợp lệ.

Các kết hợp không hợp lệ bao gồm:

|  |
| --- |
| []() - không thể có một mảng của hàm  ()() - không thể có một hàm trả về hàm  ()[] - không thể có một hàm trả về mảng |

Trong tất cả các trường hợp trên, bạn sẽ cần một tập hợp dấu ngoặc để kết nối một biểu tượng \* ở bên trái giữa các biểu tượng () và [] ở bên phải để làm cho khai báo hợp lệ.

Dưới đây là một số ví dụ khác:

**Hợp lệ:**

|  |
| --- |
| int i; an int  int \*p; an int pointer (ptr to an *int*)  int a[]; an array of ints  int f(); a function returning an int  int \*\*pp; a pointer to an int pointer (ptr to a ptr to an *int*)  int (\*pa)[]; a pointer to an array of ints  int (\*pf)(); a pointer to a function returning an int  int \*ap[]; an array of int pointers (array of ptrs to *ints*)  int aa[][]; an array of arrays of ints  int \*fp(); a function returning an int pointer  int \*\*\*ppp; a pointer to a pointer to an int pointer  int (\*\*ppa)[]; a pointer to a pointer to an array of ints  int (\*\*ppf)(); a pointer to a pointer to a function returning an int  int \*(\*pap)[]; a pointer to an array of int pointers  int (\*paa)[][]; a pointer to an array of arrays of ints  int \*(\*pfp)(); a pointer to a function returning an int pointer  int \*\*app[]; an array of pointers to int pointers  int (\*apa[])[]; an array of pointers to arrays of ints  int (\*apf[])(); an array of pointers to functions returning an int  int \*aap[][]; an array of arrays of int pointers  int aaa[][][]; an array of arrays of arrays of int  int \*\*fpp(); a function returning a pointer to an int pointer  int (\*fpa())[]; a function returning a pointer to an array of ints  int (\*fpf())(); a function returning a pointer to a function returning an int |

**Không hợp lệ:**

|  |
| --- |
| int af[](); an array of functions returning an int  int fa()[]; a function returning an array of ints  int ff()(); a function returning a function returning an int  int (\*pfa)()[]; a pointer to a function returning an array of ints  int aaf[][](); an array of arrays of functions returning an int  int (\*paf)[](); a pointer to a an array of functions returning an int  int (\*pff)()(); a pointer to a function returning a function returning an int  int \*afp[](); an array of functions returning int pointers  int afa[]()[]; an array of functions returning an array of ints  int aff[]()(); an array of functions returning functions returning an int  int \*fap()[]; a function returning an array of int pointers  int faa()[][]; a function returning an array of arrays of ints  int faf()[](); a function returning an array of functions returning an int  int \*ffp()(); a function returning a function returning an int pointer |

Nguồn: <http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html>

# STRUCTURE PADDING VÀ PACKING

Mặc định, trình biên dịch C sắp xếp cấu trúc sao cho mỗi thành viên có thể được truy cập nhanh chóng, mà không phải chịu **incurring penalties** cho **'unaligned access’**, một vấn đề với các máy RISC như DEC Alpha và một số CPU ARM.

Tùy thuộc vào kiến trúc CPU và trình biên dịch, một cấu trúc có thể chiếm nhiều không gian trong bộ nhớ hơn tổng kích thước của các thành viên thành phần của nó. Trình biên dịch có thể thêm padding (phần đệm) giữa các **members** hoặc ở cuối cấu trúc, nhưng không phải ở đầu.

**Packing** sẽ ghi đè lớp **padding** mặc định.

## PACKING STRUCTURES

Mặc định, cấu trúc được **padded** trong C. Nếu bạn muốn tránh hành vi này, bạn phải yêu cầu nó một cách rõ ràng. Dưới GCC, đó là **attribute((packed))**. Xem ví dụ này trên máy 64 bit:

|  |
| --- |
| struct *foo* {  char \*p; /\* 8 byte \*/  char c; /\* 1 byte \*/  long x; /\* 8 byte \*/  }; |

Cấu trúc này sẽ tự động được **padded** để có **8-byte alignment** và sẽ trông như sau:

|  |
| --- |
| struct *foo* {  char \*p; /\* 8 byte \*/  char c; /\* 1 byte \*/  char pad[7]; /\* 7 byte được thêm vào bởi trình biên dịch \*/  long x; /\* 8 byte \*/  }; |

Vì vậy, **sizeof (struct foo**) sẽ cho chúng ta kết quả là 24 thay vì 17. Điều này xảy ra vì trình biên dịch 64 bit đọc/ghi từ/bộ nhớ trong **8 byte của word** trong mỗi bước và rõ ràng khi cố gắng viết **char c**; một byte vào bộ nhớ thì 8 byte hoàn chỉnh (tức là **word**) được đọc và chỉ tiêu tốn **byte đầu tiên** của nó và **bảy byte tiếp theo** của nó không được sử dụng và không thể truy cập cho bất kỳ hoạt động đọc và ghi nào cho việc **structure padding**.

**Structure packing**

Nhưng nếu bạn thêm thuộc tính **packed**, trình biên dịch sẽ không thêm **padding**:

|  |
| --- |
| struct \_\_attribute\_\_((\_\_packed\_\_)) *foo* {  char \*p; /\* 8 byte \*/  char c; /\* 1 byte \*/  long x; /\* 8 byte \*/  }; |

Bây giờ **sizeof(struct foo)** sẽ trả về 17.

Thường, **packed structures** được sử dụng để:

* Để tiết kiệm không gian.
* Để định dạng cấu trúc dữ liệu để truyền qua mạng mà không phụ thuộc vào **architecture alignment** của từng nút của mạng.

Cần phải lưu ý rằng một số bộ xử lý như ARM Cortex-M0 không cho phép truy cập bộ nhớ **unaligned memory**; trong những trường hợp như vậy, **structure packing** có thể dẫn đến hành vi không xác định và có thể làm hỏng CPU.

## STRUCTURE PADDING

Giả sử cấu trúc này được định nghĩa và biên dịch bằng trình biên dịch 32 bit:

|  |
| --- |
| struct *test\_32* {  int a; // 4 byte  short b; // 2 byte  int c; // 4 byte  } str\_32; |

Chúng ta có thể mong đợi cấu trúc này chiếm 10 byte bộ nhớ, nhưng khi in ra sizeof (str\_32) thì chúng ta thấy nó sử dụng 12 byte.

Điều này xảy ra vì trình biên dịch **aligns variables** để truy cập nhanh. Một mẫu phổ biến là khi **kiểu cơ sở** chiếm N byte (trong đó N là một lũy thừa của 2 như 1, 2, 4, 8, 16 - và hiếm khi có kích thước lớn hơn), biến nên được **aligned** trên ranh giới N-byte (là bội số của N byte).

Cho cấu trúc được hiển thị với **sizeof(int) == 4** và **sizeof(short) == 2**, một bố trí phổ biến là:

|  |
| --- |
| int a; lưu trữ tại vị trí offset 0; kích thước 4.  short b; lưu trữ tại vị trí offset 4; kích thước 2.  khoảng trống không tên tại vị trí offset 6; kích thước 2.  int c; lưu trữ tại vị trí offset 8; kích thước 4. |

|  |  |  |  |
| --- | --- | --- | --- |
| 11 | 10 | 9 | 8 |
| **c** | **c** | **c** | **c** |
| 7 | 6 | 5 | 4 |
| **padding** | **padding** | **b** | **b** |
| 3 | 2 | 1 | 0 |
| **a** | **a** | **a** | **a** |

Do đó, struct test\_32 chiếm 12 byte bộ nhớ. Trong ví dụ này, không có **trailing padding**.

Trình biên dịch sẽ đảm bảo rằng các biến struct test\_32 được lưu trữ bắt đầu từ ranh giới 4 byte, để các thành viên trong cấu trúc sẽ được **properly aligned** cho việc truy cập nhanh. Các hàm phân bổ bộ nhớ như malloc(), calloc() và realloc() phải đảm bảo rằng con trỏ được trả về có **sufficiently well aligned** cho việc sử dụng với bất kỳ kiểu dữ liệu nào, vì vậy các cấu trúc được cấp phát động cũng sẽ được **aligned** đúng.

Bạn có thể gặp tình huống kỳ lạ như trên bộ xử lý Intel x86\_64 64 bit (ví dụ Intel Core i7 - một máy Mac chạy macOS Sierra hoặc Mac OS X), khi biên dịch ở chế độ 32 bit, trình biên dịch đặt **double aligned on a 4-byte boundary**; nhưng trên cùng phần cứng, khi biên dịch ở chế độ 64 bit, trình biên dịch đặt **double aligned on an 8- byte boundary**.

# MEMORY MANAGEMENT (QUẢN LÝ BỘ NHỚ)

|  |  |
| --- | --- |
| **Tên** | **sự miêu tả** |
| size (malloc, realloc và aligned\_alloc) | Kích thước tổng cộng của bộ nhớ theo byte. Đối với aligned\_alloc, kích thước phải là bội số nguyên của alignment. |
| size (calloc) | kích thước của mỗi phần tử |
| nelements | Số lượng các phần tử |
| ptr | Con trỏ trỏ đến bộ nhớ được cấp phát trước đó bởi malloc, calloc, realloc hoặc aligned\_alloc. |
| alignment | Alignment bộ nhớ được cấp phát |

Để quản lý bộ nhớ được cấp phát động, thư viện tiêu chuẩn C cung cấp các hàm malloc(), calloc(), realloc() và free(). Trong C99 và phiên bản sau, còn có aligned\_alloc(). Một số hệ thống cũng cung cấp alloca().

## ALLOCATING MEMORY (CẤP PHÁT BỘ NHỚ)

### Cấp phát tiêu chuẩn

Các hàm cấp phát bộ nhớ động của C được định nghĩa trong tiêu đề <stdlib.h>. Nếu bạn muốn cấp phát không gian bộ nhớ cho một đối tượng một cách động, bạn có thể sử dụng mã sau:

|  |
| --- |
| int \*p = malloc(10 \* sizeof \**p*);  if (p == NULL)  {  perror("malloc() failed");  return -1;  } |

Điều này tính toán số byte mà mười **int** chiếm trong bộ nhớ, sau đó yêu cầu nhiều byte đó từ **malloc** và gán kết quả (nghĩa là địa chỉ bắt đầu của đoạn bộ nhớ vừa được tạo bằng **malloc**) cho một con trỏ có tên p

Việc sử dụng sizeof để tính toán lượng bộ nhớ cần yêu cầu là việc tốt vì kết quả của sizeof được định nghĩa bởi người triển khai (trừ các kiểu ký tự, đó là char, signed char và unsigned char, với sizeof được định nghĩa luôn luôn là 1).

**Vì malloc có thể không đáp ứng được yêu cầu nên nó có thể trả về một con null. Điều quan trọng là phải kiểm tra điều này để ngăn chặn các nỗ lực dereference con trỏ null sau này.**

Bộ nhớ được cấp phát động bằng **malloc()** có thể được thay đổi kích thước bằng **realloc()** hoặc, khi không còn cần thiết, được giải phóng bằng **free().**

Hoặc, việc khai báo int array[10]; sẽ cấp phát cùng lượng bộ nhớ. Tuy nhiên, nếu nó được khai báo bên trong một hàm mà không có từ khóa static, nó chỉ có thể sử dụng trong phạm vi của hàm mà nó được khai báo và các hàm mà nó gọi (bởi vì mảng sẽ được cấp phát trên ngăn xếp và không gian sẽ được giải phóng để tái sử dụng khi hàm trả về). Hoặc, nếu nó được định nghĩa với static trong một hàm, hoặc nếu nó được định nghĩa bên ngoài bất kỳ hàm nào, thì tuổi thọ của nó là tuổi thọ của chương trình. Con trỏ cũng có thể được trả về từ một hàm, tuy nhiên một hàm trong C không thể trả về một mảng.

### Zeroed Memory

Bộ nhớ được trả về bởi malloc có thể không được khởi tạo thành giá trị hợp lý, và cần phải chú ý để đặt giá trị 0 vào bộ nhớ với memset hoặc sao chép ngay lập tức một giá trị thích hợp vào nó. Hoặc, calloc trả về một khối bộ nhớ với kích thước mong muốn trong đó tất cả các bit được khởi tạo thành 0. Điều này không cần phải giống như biểu diễn của số 0 dấu phẩy động hoặc hằng số con trỏ null.

|  |
| --- |
| int \*p = calloc(10, sizeof \**p*);  if (p == NULL)  {  perror("calloc() failed");  return -1;  } |

Một lưu ý về **calloc**: Hầu hết các triển khai (phổ biến) sẽ tối ưu hóa **calloc()** cho hiệu suất, vì vậy nó sẽ **nhanh hơn** việc gọi **malloc()**, sau đó **memset()**, mặc dù hiệu ứng tông rất giống nhau.

### Aligned Memory

Phiên bản ≥ C11

C11 giới thiệu một hàm mới là **aligned\_alloc()** có khả năng cấp phát không gian với **alignment** đã cho. Nó có thể được sử dụng nếu bộ nhớ cần được cấp phát cần được aligned tại các ranh giới cụ thể không thể được đáp ứng bởi **malloc()** hoặc **calloc().** Các hàm malloc() và calloc() cấp phát bộ nhớ **aligned** cho bất kỳ kiểu đối tượng nào (tức là sự căn chỉnh là alignof(max\_align\_t)). Nhưng với aligned\_alloc(), có thể yêu cầu các **alignments** lớn hơn.

|  |
| --- |
| /\* Allocates 1024 bytes with 256 bytes alignment. \*/  char \*ptr = aligned\_alloc(256, 1024);  if (ptr) {  perror("aligned\_alloc()");  return -1;  }  free(ptr); |

Tiêu chuẩn C11 đặt hai hạn chế:

1) kích thước (đối số thứ hai) yêu cầu phải là bội số nguyên của alignment (đối số đầu tiên) và

2) giá trị của alignment phải là alignment hợp lệ được hỗ trợ bởi triển khai. Vi phạm một trong hai điều này dẫn đến hành vi không xác định.

## GIẢI PHÓNG BỘ NHỚ (FREEING MEMORY)

Có thể giải phóng bộ nhớ được cấp phát động bằng cách gọi hàm free().

|  |
| --- |
| int \*p = malloc(10 \* sizeof \**p*); /\* cấp phát bộ nhớ \*/  if (p == NULL)  {  perror("malloc failed");  return -1;  }  free(p); /\* giải phóng bộ nhớ \*/  /\* chú ý rằng sau khi free(p), thậm chí việc sử dụng \*giá trị\* của con trỏ p  có hành vi không xác định, cho đến khi một giá trị mới được lưu trữ vào nó. \*/  /\* tái sử dụng/tái mục đích chính con trỏ \*/  int i = 42;  p = &i; /\* Điều này hợp lệ, có hành vi xác định \*/ |

Bộ nhớ mà con trỏ p trỏ đến sẽ được thu hồi (hoặc bởi triển khai libc hoặc bởi hệ điều hành dưới), do đó **truy cập vào khối bộ nhớ bị giải phóng** thông qua p sẽ dẫn đến hành vi không xác định. Con trỏ trỏ đến các yếu tố bộ nhớ đã được giải phóng thường được gọi là con trỏ treo (dangling pointers), và tạo ra một rủi ro về bảo mật.

Hơn nữa, tiêu chuẩn C nêu rõ rằng thậm chí việc truy cập giá trị của con trỏ treo có hành vi không xác định.

Chú ý rằng con trỏ p có thể được tái sử dụng chính nó như được hiển thị ở trên.

Vui lòng lưu ý rằng bạn chỉ có thể gọi free() trên các con trỏ đã được trả về trực tiếp từ các hàm malloc(), calloc(), realloc() và aligned\_alloc(), hoặc khi tài liệu cho bạn biết bộ nhớ đã được cấp phát theo cách đó (các hàm như strdup () là các ví dụ đáng chú ý). Việc giải phóng một con trỏ được, được đạt được bằng cách sử dụng toán tử & trên một biến, hoặc nằm ở giữa một khối được cấp phát, là bị cấm. Lỗi như vậy thường không được chẩn đoán bởi trình biên dịch của bạn nhưng sẽ dẫn đến trạng thái không xác định của chương trình.

Có hai phương pháp phổ biến để ngăn các trường hợp không xác định như vậy.

**Phương pháp đầu tiên và ưu tiên là đơn giản - khi p không còn cần thiết, p chính nó ngưng tồn tại, ví dụ:**

|  |
| --- |
| if (something\_is\_needed())  {  int \*p = malloc(10 \* sizeof \*p);  if (p == NULL)  {  perror("malloc failed");  return -1;  }  /\* thực hiện bất kỳ điều gì cần với p \*/  free(p);  } |

Bằng cách gọi free() trực tiếp trước khi kết thúc khối chứa (tức là }), p chính nó ngưng tồn tại. Trình biên dịch sẽ hiển thị một lỗi biên dịch trên mọi nỗ lực để sử dụng p sau đó.

**Phương pháp thứ hai là vô hiệu hóa con trỏ chính nó sau khi giải phóng bộ nhớ mà nó trỏ đến:**

|  |
| --- |
| free(p);  p = NULL; // bạn cũng có thể sử dụng 0 thay vì NULL |

Lý do cho phương pháp này:

* Trên nhiều nền tảng, việc cố gắng dereference một con trỏ null sẽ gây ra sự cố ngay lập tức: Segmentation fault. Ở đây, ít nhất chúng ta có một stack trace trỏ đến biến đã được sử dụng sau khi được giải phóng.
* Không đặt con trỏ thành NULL, chúng ta có con trỏ treo. Chương trình có khả năng sẽ vẫn bị lỗi, nhưng sau đó, vì bộ nhớ mà con trỏ trỏ tới sẽ âm thầm bị hỏng. Lỗi như vậy khó để theo dõi vì chúng có thể dẫn đến một stack gọi hoàn toàn không liên quan đến vấn đề ban đầu. Phương pháp này tuân theo nguyên tắc "fail-fast" (sự thất bại nhanh chóng).
* Giải phóng một con trỏ null là an toàn. Tiêu chuẩn C xác định rằng free(NULL) không có tác dụng:
  + Hàm free gây ra việc không gian mà ptr trỏ đến được giải phóng, nghĩa là được thực hiện cho phép cấp phát tiếp theo. Nếu ptr là một con trỏ null, không có hành động nào xảy ra. Nếu
  + đối số không phù hợp với một con trỏ trước đó được trả về bởi hàm calloc, malloc hoặc realloc, hoặc nếu không gian đã được giải phóng bởi cuộc gọi free hoặc realloc, hành vi là không xác định.
* Đôi khi, phương pháp đầu tiên không thể được sử dụng (ví dụ: bộ nhớ được cấp phát trong một hàm và được giải phóng sau đó trong một hàm hoàn toàn khác).

## THAY ĐỔI KÍCH THƯỚC BỘ NHỚ CẤP PHÁT (REALLOCATING MEMORY)

Có thể bạn cần mở rộng hoặc thu hẹp không gian lưu trữ của con trỏ sau khi bạn đã cấp phát bộ nhớ cho nó. Hàm void \*realloc(void \*ptr, size\_t size) giải phóng đối tượng cũ được trỏ bởi ptr và trả về một con trỏ đến một đối tượng có kích thước được xác định bởi size. ptr là con trỏ đến một khối bộ nhớ được cấp phát trước đó bằng malloc, calloc hoặc realloc (hoặc con trỏ null) để được cấp phát lại. Nội dung tối đa có thể của bộ nhớ gốc được bảo tồn. Nếu kích thước mới lớn hơn, bất kỳ bộ nhớ bổ sung vượt quá kích thước cũ sẽ không được khởi tạo. Nếu kích thước mới ngắn hơn, nội dung của phần thu nhỏ bị mất. Nếu ptr là NULL, một khối mới được cấp phát và một con trỏ đến nó được trả về bởi hàm.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(void)  {  int \*p = malloc(10 \* sizeof \*p);  if (NULL == p)  {  perror("malloc() failed");  return EXIT\_FAILURE;  }  p[0] = 42;  p[9] = 15;  /\* Reallocate array to a larger size, storing the result into a  \* temporary pointer in case realloc() fails. \*/  {  int \*temporary = realloc(p, 1000000 \* sizeof \*temporary);  /\* realloc() failed, the original allocation was not free'd yet. \*/  if (NULL == temporary)  {  perror("realloc() failed");  free(p); /\* Clean up. \*/  return EXIT\_FAILURE;  }  p = temporary;  }  /\* From here on, array can be used with the new size it was  \* realloc'ed to, until it is free'd. \*/  /\* The values of p[0] to p[9] are preserved, so this will print:  42 15  \*/  printf("%d %d\n", p[0], p[9]);  free(p);  return EXIT\_SUCCESS;  } |

Đối tượng đã được cấp phát lại có thể hoặc không có địa chỉ giống như \*p. Do đó, quan trọng là bắt giá trị trả về từ realloc chứa địa chỉ mới nếu lời gọi thành công.

Hãy đảm bảo bạn gán giá trị trả về của realloc vào một biến tạm thay vì p ban đầu. realloc sẽ trả về null trong trường hợp bất kỳ sự cố nào, điều này sẽ ghi đè lên con trỏ. Điều này sẽ làm mất dữ liệu của bạn và tạo ra một memory leak.

## REALLOC(PTR, 0) KHÔNG TƯƠNG ĐƯƠNG VỚI FREE(PTR)

realloc cơ bản tương đương với việc malloc + memcpy + free trên con trỏ khác.

Nếu kích thước không gian yêu cầu là 0, hành vi của realloc là xác định bởi cài đặt. Điều này tương tự cho tất cả các hàm cấp phát bộ nhớ nhận tham số kích thước có giá trị 0. Những hàm như vậy thực tế có thể trả về một con trỏ không phải là null, nhưng con trỏ đó không bao giờ được dereferenced.

Do đó, realloc(ptr, 0) không tương đương với free(ptr). Nó có thể:

* Là một cài đặt "lười" và chỉ trả về ptr.
* Giải phóng ptr, cấp phát một phần tử giả và trả về nó.
* Giải phóng ptr và trả về 0.
* Chỉ trả về 0 cho thất bại và không làm gì khác.

Vì vậy, đặc biệt là hai trường hợp cuối cùng không thể phân biệt được bằng mã ứng dụng.

Điều này có nghĩa là realloc(ptr, 0) có thể không thực sự giải phóng/bộ nhớ, và do đó nó không bao giờ nên được sử dụng để thay thế cho free.

## MẢNG ĐA CHIỀU CÓ KÍCH THƯỚC THAY ĐỔI

Phiên bản ≥ C99

Kể từ C99, C hỗ trợ mảng có kích thước biến đổi (VLA), đại diện cho các mảng có giới hạn chỉ được biết vào thời điểm khởi tạo. Mặc dù bạn cần phải cẩn thận để không cấp phát VLA quá lớn (chúng có thể gây sự cố cho stack của bạn), sử dụng con trỏ đến VLA và sử dụng chúng trong biểu thức sizeof là hoàn toàn hợp lệ.

|  |
| --- |
| double sumAll(size\_t *n*, size\_t *m*, double *A*[*n*][*m*]) {  double ret = 0.0;  for (size\_t i = 0; i < *n*; ++i)  for (size\_t j = 0; j < *m*; ++j)  ret += *A*[i][j]  return ret;  }  int main(int *argc*, char \**argv*[*argc*+1]) {  size\_t n = *argc*\*10;  size\_t m = *argc*\*8;  double (\*matrix)[m] = malloc(sizeof(double[n][m]));  // khởi tạo ma trận bằng cách nào đó  double res = sumAll(n, m, matrix);  printf("result is %g\n", res);  free(matrix);  } |

Ở đây, matrix là một con trỏ đến các phần tử kiểu double[m], và biểu thức sizeof với double[n][m] đảm bảo rằng nó chứa không gian cho n phần tử như vậy. Tất cả không gian này được cấp phát liên tiếp và do đó có thể bị giải phóng bằng một lời gọi đơn cho hàm free.

Sự xuất hiện của VLA trong ngôn ngữ cũng ảnh hưởng đến các khai báo có thể của mảng và con trỏ trong tiêu đề hàm. Bây giờ, một biểu thức số nguyên tổng quát được phép ở bên trong dấu [] của các tham số mảng. Cho cả hai hàm, các biểu thức trong [] sử dụng các tham số đã được khai báo trước trong danh sách tham số. Đối với hàm sumAll, đó là các độ dài mà mã nguồn của người dùng mong đợi cho ma trận. Như với tất cả các tham số hàm mảng trong C, chiều trong cùng được viết lại thành một kiểu con trỏ, vì vậy điều này tương đương với khai báo:

|  |
| --- |
| double sumAll(size\_t *n*, size\_t *m*, double (\**A*)[*m*]); |

Tức là, n thực sự không phải là một phần của giao diện hàm, nhưng thông tin này có thể hữu ích cho tài liệu và nó cũng có thể được sử dụng bởi các trình biên dịch kiểm tra giới hạn để cảnh báo về truy cập vượt giới hạn.

Tương tự, đối với hàm main, biểu thức argc+1 là độ dài tối thiểu mà tiêu chuẩn C quy định cho đối số argv.

Lưu ý rằng chính thức hỗ trợ VLA là tùy chọn trong C11, nhưng chúng ta không biết có trình biên dịch nào thực hiện C11 mà không có chúng. Bạn có thể kiểm tra bằng macro \_\_STDC\_NO\_VLA\_\_ nếu cần.

## ALLOCA: CẤP PHÁT BỘ NHỚ TRÊN STACK

Cảnh báo: alloca chỉ được đề cập ở đây để hoàn thiện. Nó hoàn toàn không có khả năng chuyển đổi (không được bao gồm trong bất kỳ tiêu chuẩn thông thường nào) và có một số tính năng tiềm ẩn nguy hiểm có thể làm cho nó không an toàn cho những người không có kinh nghiệm. Mã C hiện đại nên thay thế nó bằng Variable Length Arrays (VLA).

Manual page

|  |
| --- |
| #include <alloca.h>  // phiên bản glibc của stdlib.h mặc định bao gồm alloca.h  void foo(int *size*) {  char \*data = alloca(*size*);  /\*  thân hàm;  \*/  // dữ liệu tự động được giải phóng  } |

Cấp phát bộ nhớ trên stack frame của hàm gọi, không gian được trỏ đến bởi con trỏ trả về sẽ tự động được giải phóng khi hàm gọi kết thúc.

Mặc dù hàm này thuận tiện cho việc quản lý bộ nhớ tự động, hãy nhớ rằng yêu cầu cấp phát lớn có thể gây ra stack overflow và bạn không thể sử dụng free với bộ nhớ được cấp phát bằng alloca (điều này có thể gây ra vấn đề hơn với stack overflow).

Vì những lý do này, không nên sử dụng alloca trong một vòng lặp hoặc trong một hàm đệ quy.

Và vì bộ nhớ được giải phóng khi hàm trả về, bạn không thể trả con trỏ như một kết quả của hàm (hành vi này sẽ không xác định).

**Tóm lại**

* Gọi giống với malloc.
* Tự động được giải phóng khi hàm trả về.
* Không tương thích với các hàm free và realloc (hành vi không xác định).
* Con trỏ không thể được trả lại như kết quả của một hàm (hành vi không xác định).
* Kích thước cấp phát giới hạn bởi không gian stack, mà (trên hầu hết các máy tính) nhỏ hơn nhiều so với không gian heap có sẵn để sử dụng bởi malloc.
* Tránh sử dụng alloca và VLA (mảng kích thước biến đổi) trong một hàm duy nhất.

**Khuyến khích**

* Không nên sử dụng **alloca( )** trong mã mới.

Phiên bản ≥ C99:

Thay thế hiện đại.

|  |
| --- |
| void foo(int *size*) {  char data[*size*];  /\*  thân hàm;  \*/  // dữ liệu tự động được giải phóng  } |

Điều này hoạt động tương tự như alloca, và hoạt động ở những nơi mà alloca không hoạt động (trong vòng lặp, ví dụ). Nó giả định một trong hai: hiện thực C99 hoặc hiện thực C11 không xác định \_\_STDC\_NO\_VLA\_\_.

## QUẢN LÝ BỘ NHỚ DO NGƯỜI DÙNG ĐỊNH NGHĨA

Hàm malloc() thường gọi các hàm của hệ thống điều hành để thu được các trang bộ nhớ. Nhưng không có gì đặc biệt về hàm này và nó có thể được triển khai trực tiếp bằng ngôn ngữ C bằng cách khai báo một mảng tĩnh lớn và cấp phát từ nó (có một chút khó khăn trong việc đảm bảo căn chỉnh đúng, trong thực tế căn chỉnh đến 8 byte hầu như luôn luôn là đủ).

Để triển khai một kế hoạch đơn giản, một khối kiểm soát được lưu trữ trong vùng bộ nhớ ngay trước con trỏ sẽ được trả về từ cuộc gọi. Điều này có nghĩa là free() có thể được triển khai bằng cách trừ đi từ con trỏ trả về và đọc thông tin kiểm soát, thông tin này thường là kích thước khối cộng thêm một số thông tin cho phép đặt nó trở lại danh sách không cấp phát - một danh sách liên kết của các khối chưa cấp phát.

Khi người dùng yêu cầu cấp phát, danh sách không cấp phát được tìm kiếm cho đến khi tìm thấy một khối có kích thước giống hoặc lớn hơn số lượng được yêu cầu, sau đó nếu cần, khối sẽ được chia. Điều này có thể dẫn đến sự mảng bộ nhớ nếu người dùng liên tục thực hiện nhiều cấp phát và giải phóng không dự đoán được về kích thước và tại các khoảng thời gian không dự đoán (không phải tất cả các chương trình thực sự hoạt động như vậy, kế hoạch đơn giản thường là đủ cho các chương trình nhỏ).

|  |
| --- |
| /\* khối kiểm soát tiêu biểu \*/  struct *block*  {  size\_t size; /\* kích thước của khối \*/  struct *block* \*next; /\* khối kế tiếp trong danh sách không cấp phát \*/  struct *block* \*prev; /\* con trỏ phía trước đến khối trước trong bộ nhớ \*/  void \*padding; /\* cần 16 byte để làm cho nhiều lần của 8 \*/  }  static struct *block* arena[10000]; /\* cấp phát từ đây \*/  static struct *block* \*firstfree; |

Nhiều chương trình yêu cầu số lượng lớn cấp phát các đối tượng nhỏ có cùng kích thước. Điều này rất dễ triển khai. Đơn giản sử dụng một khối với con trỏ `next`. Ví dụ, nếu cần một khối 32 byte:

|  |
| --- |
| union block  {  union block \* next;  unsigned char payload[32];  }  static union block arena[100];  static union block \* head;  void init(void)  {  int i;  for (i = 0; i < 100 - 1; i++)  arena[i].next = &arena[i + 1];  arena[i].next = 0; /\* cuối cùng, null \*/  head = &block[0];  }  void \*block\_alloc()  {  void \*answer = head;  if (answer)  head = head->next;  return answer;  }  void block\_free(void \**ptr*)  {  union block \*block = *ptr*;  block->next = head;  head - block;  } |

Kế hoạch này vô cùng nhanh và hiệu quả và có thể trở nên thông dụng với việc mất một phần sự rõ ràng.

# THỰC HIỆN HÀNH VI ĐƯỢC XÁC ĐỊNH (IMPLEMENTATION – DEFINED BEHAVIOUR)

## DỊCH PHẢI MỘT SỐ NGUYÊN ÂM

|  |
| --- |
| int signed\_integer = -1;  // The right shift operation exhibits implementation-defined behavior:  int result = signed\_integer >> 1; |

## GÁN MỘT GIÁ TRỊ NẰM NGOÀI PHẠM VI CỦA MỘT SỐ NGUYÊN

|  |
| --- |
| // Supposing SCHAR\_MAX, the maximum value that can be represented by a signed char, is  // 127, the behavior of this assignment is implementation-defined:  signed char integer;  integer = 128; |

## CẤP PHÁT SỐ BYTE BẰNG 0

|  |
| --- |
| // The allocation functions have implementation-defined behavior when the requested size  // of the allocation is zero.  void \*p = malloc(0); |

## BIỂU DIỄN SỐ NGUYÊN CÓ DẤU

Mỗi loại số nguyên đã ký có thể được biểu diễn ở bất kỳ một trong ba định dạng nào; nó được xác định theo triển khai cái nào được sử dụng. Việc triển khai được sử dụng cho bất kỳ loại số nguyên đã ký nào có độ rộng tối thiểu intcó thể được xác định trong thời gian chạy từ hai bit có thứ tự thấp nhất của biểu diễn giá trị -1trong loại đó, như vậy:

|  |
| --- |
| enum { sign\_magnitude = 1, ones\_compl = 2, twos\_compl = 3, };  #define SIGN\_REP(*T*) ((T)-1 & (T)3)  switch (SIGN\_REP(long)) {  case sign\_magnitude: { /\* làm gì đó \*/ break; }  case ones\_compl: { /\* làm khác đi \*/ break; }  case twos\_compl: { /\* làm khác nữa \*/ break; }  case 0: { \_Static\_assert(SIGN\_REP(long), "biểu diễn dấu giả"); }  } |

Mẫu tương tự áp dụng cho việc biểu diễn các loại hẹp hơn, nhưng chúng không thể được kiểm tra bằng kỹ thuật này vì các toán hạng của &phải tuân theo "các phép chuyển đổi số học thông thường" trước khi tính toán kết quả.

# ATOMICS

## ATOMICS VÀ CÁC TOÁN TỬ

Biến atomics có thể được truy cập cùng lúc bởi các luồng khác nhau mà không tạo ra tình trạng cạnh tranh (race conditions).

|  |
| --- |
| /\* a global static variable that is visible by all threads \*/  static unsigned \_Atomic active = ATOMIC\_VAR\_INIT(0);  int myThread(void\* *a*) {  ++active; // increment active race free  // do something  --active; // decrement active race free  return 0;  } |

Tất cả các hoạt động lvalue (hoạt động sửa đổi đối tượng) cho phép cho kiểu cơ bản đều được cho phép và sẽ không dẫn đến tình trạng cạnh tranh giữa các luồng khác nhau truy cập chúng.

* Các hoạt động trên các đối tượng atomic thường nhanh hơn hàng động so với các hoạt động số học thông thường. Điều này bao gồm cả các hoạt động load hoặc store đơn giản. Vì vậy, bạn chỉ nên sử dụng chúng cho các nhiệm vụ quan trọng.
* Các hoạt động số học thông thường và gán như a = a+1; thực sự là ba hoạt động trên a: trước hết là một hoạt động load, sau đó là phép cộng và cuối cùng là một hoạt động store. Điều này không an toàn đối với tình trạng cạnh tranh. Chỉ có các hoạt động a += 1; và a++; là an toàn.

# CÂU LỆNH NHẢY (JUMP STATEMENTS)

## SỬ DỤNG RETURN

**Trả về một giá trị**

Một trường hợp thường sử dụng: trả về từ hàm main()

|  |
| --- |
| #include <stdlib.h> /\* cho các macro EXIT\_xxx \*/  int main(int *argc*, char \*\* *argv*)  {  if (2 < *argc*)  {  return EXIT\_FAILURE; /\* Chương trình yêu cầu một đối số:  ra khỏi ngay lập tức và bỏ qua phần còn lại của mã chương trình. \*/  }  /\* Làm điều gì đó. \*/  return EXIT\_SUCCESS;  } |

Ghi chú bổ sung

* Đối với một hàm có kiểu trả về là void (không bao gồm void \* hoặc các kiểu liên quan), câu lệnh return không nên có bất kỳ biểu thức đi kèm nào; tức là, câu lệnh return duy nhất được phép là return;.
* Đối với một hàm có kiểu trả về không phải là void, câu lệnh return không được xuất hiện mà không có biểu thức.
* Đối với hàm main() (và chỉ đối với main()), một câu lệnh return tường minh không bắt buộc (trong C99 hoặc phiên bản sau đó). Nếu việc thực thi đạt tới dấu }, một giá trị ngầm định là 0 sẽ được trả về. Một số người cho rằng việc bỏ qua câu lệnh return này là tập quán không tốt; những người khác đề xuất để bỏ nó đi.

**Không trả về gì cả**

Trả về từ một hàm void

|  |
| --- |
| void log(const char \* *message\_to\_log*)  {  if (NULL == *message\_to\_log*)  {  return; /\* Không có gì để ghi nhật ký, đi về ngay bây giờ, bỏ qua việc ghi nhật ký. \*/  }  fprintf(stderr, "%s:%d %s\n", \_\_FILE\_\_, \_LINE\_\_, *message\_to\_log*);  return; /\* Tùy chọn, vì hàm này không trả về giá trị gì cả. \*/  } |

## SỬ DỤNG GOTO ĐỂ NHẢY RA KHỎI CÁC VÒNG LẶP LỒNG NHAU

|  |
| --- |
| size\_t i, j;  for (i = 0; i < myValue && !breakout\_condition; ++i) {  for (j = 0; j < mySecondValue && !breakout\_condition; ++j) {  ... /\* Làm một số việc gì đó, có thể thay đổi breakout\_condition \*/  /\* Khi breakout\_condition == true thì vòng lặp kết thúc \*/  }  } |

Nhưng ngôn ngữ C cung cấp câu lệnh goto, có thể hữu ích trong trường hợp này. Bằng cách sử dụng nó với một nhãn được khai báo sau các vòng lặp, chúng ta có thể dễ dàng thoát khỏi các vòng lặp.

|  |
| --- |
| size\_t i, j;  for (i = 0; i < myValue; ++i) {  for (j = 0; j < mySecondValue; ++j) {  ...  if(breakout\_condition)  goto final;  }  }  final: |

Tuy nhiên, thường khi nhu cầu này xuất hiện, việc sử dụng câu lệnh return có thể tốt hơn. Cấu trúc này cũng được coi là "không có cấu trúc" trong lý thuyết lập trình cấu trúc.

Tình huống khác mà goto có thể hữu ích là để chuyển đến một trình xử lý lỗi:

|  |
| --- |
| ptr = malloc(N \* *x*);  if (!ptr)  goto out\_of\_memory;  /\* xử lý bình thường \*/  free(*ptr*);  return SUCCESS;  out\_of\_memory:  free(ptr); /\* không gây hại, và cần thiết nếu chúng ta có lỗi tiếp theo \*/  return FAILURE; |

Việc sử dụng gotogiữ luồng lỗi tách biệt với luồng điều khiển chương trình thông thường. Tuy nhiên, nó cũng được coi là "không có cấu trúc" theo nghĩa kỹ thuật.

## SỬ DỤNG BREAK VÀ CONTINUE

Đọc ngay lập tức **continue** trên đầu vào **không hợp lệ** hoặc **break** theo yêu cầu của người dùng hoặc phần cuối của tệp:

|  |
| --- |
| #include <stdlib.h> /\* for EXIT\_xxx macros \*/  #include <stdio.h> /\* for printf() and getchar() \*/  #include <ctype.h> /\* for isdigit() \*/  void flush\_input\_stream(*FILE* \* *fp*);  int main(void)  {  int sum = 0;  printf("Enter digits to be summed up or 0 to exit:\n");  do  {  int c = getchar();  if (EOF == c)  {  printf("Read 'end-of-file', exiting!\n");  break;  }  if ('\n' != c)  {  flush\_input\_stream(stdin);  }  if (!isdigit(c))  {  printf("%c is not a digit! Start over!\n", c);  continue;  }  if ('0' == c)  {  printf("Exit requested.\n");  break;  }  sum += c - '0';  printf("The current sum is %d.\n", sum);  } while (1);  return EXIT\_SUCCESS;  }  void flush\_input\_stream(*FILE* \* *fp*)  {  *size\_t* i = 0;  int c;  while ((c = fgetc(*fp*)) != '\n' && c != EOF) /\* Pull all until and including the next new-line. \*/  {  ++i;  }  if (0 != i)  {  fprintf(stderr, "Flushed %zu characters from input.\n", i);  }  } |

# CREATE AND INCLUDE HEADER FILES

Trong ngôn ngữ C hiện đại, các tệp header là công cụ quan trọng và cần được thiết kế và sử dụng đúng cách. Chúng cho phép trình biên dịch kiểm tra chéo các phần của chương trình đã được biên dịch một cách độc lập.

Các tệp header khai báo các kiểu, hàm, macro v.v. cần thiết bởi các phần khác trong chương trình. Mã code sử dụng bất kỳ trong số các dịch vụ đó đều bao gồm tệp header. Mã code định nghĩa các dịch vụ đó cũng bao gồm tệp header. Điều này cho phép trình biên dịch kiểm tra xem việc **sử dụng** và **định nghĩa** có khớp nhau không.

## Giới thiệu

Có một số hướng dẫn cần tuân thủ khi tạo và sử dụng các tệp header trong một dự án C:

* ***Khả năng tái sử dụng***

Nếu một tệp header được bao gồm nhiều lần trong một đơn vị biên dịch (TU), nó không nên gây ra lỗi biên dịch.

* ***Tính tự động đóng gói***

Nếu bạn cần các dịch vụ được khai báo trong một tệp header, bạn không nên phải bao gồm bất kỳ tệp header nào khác một cách tường minh.

* ***Tối thiểu hóa***

Bạn không nên thể loại bỏ bất kỳ thông tin nào khỏi một tệp header mà không gây ra lỗi biên dịch.

* ***Bao gồm những gì bạn sử dụng (IWYU)***

Quan trọng hơn đối với C++ hơn là C, nhưng vẫn quan trọng trong C. Nếu mã code trong một TU (gọi là code.c) trực tiếp sử dụng các tính năng được khai báo bởi một tệp header (gọi là "headerA.h"), thì code.c nên #include "headerA.h" trực tiếp, ngay cả khi TU bao gồm một tệp header khác (gọi là "headerB.h") mà tạm thời bao gồm "headerA.h".

Đôi khi, có thể có lý do đủ tốt để vi phạm một hoặc nhiều hướng dẫn này, nhưng bạn nên nhận thức rằng bạn đang vi phạm quy tắc và nhận thức về hậu quả của việc làm như vậy trước khi vi phạm.

## SELF – CONTAINMENT

Các tiêu đề hiện đại self-contained, có nghĩa là một chương trình cần sử dụng các phương tiện được xác định bởi header.h có thể include header đó ( #include "header.h") và không phải lo lắng về việc liệu các header khác có cần được đưa vào trước hay không.

**Khuyến nghị: Các tệp header self-contained.**

**Quy luật lịch sử**

Lịch sử, đây đã là một chủ đề đôi chút gây tranh cãi.

Vào một thời đại nào đó, AT&T Indian Hill C Style and Coding Standards đã nêu rõ:

Các tệp header không nên lồng nhau. Phần đầu của một tệp header nên miêu tả những tệp header khác cần phải được #included để tệp header này có thể hoạt động. Trong trường hợp cực đoan, khi có một số lượng lớn các tệp header cần được included trong một số lượng lớn các tệp nguồn khác nhau, có thể đặt tất cả các #include chung trong một tệp include duy nhất.

Điều này hoàn toàn trái ngược với self-containment.

**Quy luật hiện đại**

Tuy nhiên, kể từ đó, dư luận đã có xu hướng ngược lại. Nếu một tệp nguồn cần sử dụng các phương tiện được khai báo bởi tiêu đề header.h, lập trình viên sẽ có thể viết:

|  |
| --- |
| #include "header.h" |

và (dưới điều kiện chỉ cần đặt các đường dẫn tìm kiếm đúng trên command line), bất kỳ tệp header tiên quyết cần thiết nào sẽ được bao gồm bởi header.h mà không cần thêm thêm tệp header nào vào source file.

Điều này cung cấp tính linh hoạt tốt hơn cho mã nguồn. Điều này cũng bảo vệ mã nguồn khỏi vấn đề "guess why this header was added" mà phát sinh sau khi mã nguồn đã được sửa đổi và sử dụng trong một hoặc hai thập kỷ .

NASA Goddard Space Flight Center (GSFC) coding standards for C là một trong những quy chuẩn hiện đại hơn - nhưng hiện đã khá khó để theo dõi. Nó nêu rõ rằng các tệp header nên được self-contained. Nó cũng cung cấp một cách đơn giản để đảm bảo rằng các tệp header được self-contained: tệp nguồn thực thi cho tệp header nên đặt tệp header làm tệp header đầu tiên. Nếu nó không self-contained, mã nguồn sẽ không biên dịch.

Lý do được GSFC đưa ra bao gồm:

§2.1.1 Giải thích về việc header include

Tiêu chuẩn này yêu cầu tệp header của một đơn vị chứa các câu lệnh #include cho tất cả các tệp header khác cần thiết bởi tệp đơn vị. Đặt #include cho tệp header đầu tiên trong phần thân đơn vị cho phép trình biên dịch xác minh rằng tệp header chứa tất cả các câu lệnh #include cần thiết.

Một thiết kế thay thế, không được phép bởi tiêu chuẩn này, không cho phép tệp header chứa bất kỳ câu lệnh #include nào; tất cả các #include được thực hiện trong các tệp nguồn. Sau đó, tệp header đơn vị phải chứa câu lệnh #ifdef kiểm tra xem các tệp header cần thiết đã được bao gồm theo đúng thứ tự hay chưa.

Một ưu điểm của thiết kế thay thế là danh sách #include trong tệp body chính xác là danh sách phụ thuộc cần thiết trong một tệp makefile và danh sách này được kiểm tra bởi trình biên dịch. Với thiết kế tiêu chuẩn, một công cụ phải được sử dụng để tạo danh sách phụ thuộc. Tuy nhiên, tất cả các môi trường phát triển được khuyến nghị của nhánh đều cung cấp một công cụ như vậy.

Một hạn chế lớn của thiết kế thay thế là nếu danh sách tệp cần thiết cho đơn vị thay đổi, mỗi tệp sử dụng đơn vị đó phải được chỉnh sửa để cập nhật danh sách câu lệnh #include. Ngoài ra, danh sách tệp cần thiết cho một thư viện trình biên dịch có thể khác nhau trên các mục tiêu khác nhau.

Một hạn chế khác của thiết kế thay thế là các tệp header của thư viện trình biên dịch và các tệp bên thứ ba khác phải được sửa đổi để thêm các câu lệnh #ifdef cần thiết.

Do đó, self-containment có nghĩa là:

* Nếu một tệp header header.h cần một tệp header lồng vào mới extra.h, bạn không cần kiểm tra tất cả các tệp nguồn sử dụng header.h để xem liệu bạn cần phải thêm extra.h không.
* Nếu một tệp header header.h không còn cần phải bao gồm một tệp header cụ thể nào đó notneeded.h, bạn không cần phải kiểm tra tất cả các tệp nguồn sử dụng header.h để xem liệu bạn có thể an toàn loại bỏ notneeded.h không (nhưng hãy xem phần "Bao gồm những gì bạn sử dụng").
* Bạn không cần thiết lập chuỗi đúng cho việc bao gồm các tệp header tiên quyết (điều này yêu cầu một sắp xếp thứ tự đỉnh để thực hiện công việc một cách chính xác).

**Kiểm tra self-containment**

Xem "Linking against a static library" để biết về tập lệnh **chkhdr** có thể được sử dụng để kiểm tra khả năng tái sử dụng và self-containment của tệp header.

## TỐI GIẢN

Headers là một cơ chế kiểm tra tính nhất quán quan trọng, nhưng chúng phải càng nhỏ càng tốt. Cụ thể, điều đó có nghĩa là một tiêu đề không nên bao gồm các tiêu đề khác chỉ vì tệp triển khai sẽ cần các tiêu đề khác. Tiêu đề chỉ nên chứa những tiêu đề cần thiết cho người tiêu dùng dịch vụ được mô tả.

Ví dụ: không nên bao gồm tiêu đề dự án <stdio.h>trừ khi một trong các giao diện chức năng sử dụng loại FILE \*(hoặc một trong các loại khác chỉ được xác định trong <stdio.h>). Nếu một giao diện sử dụng size\_t, tiêu đề nhỏ nhất đủ là <stddef.h>. Rõ ràng, nếu một tiêu đề xác định khác size\_tđược bao gồm, thì không cần phải bao gồm <stddef.h>nữa.

Nếu các headers là tối thiểu, thì nó cũng giữ thời gian biên dịch ở mức tối thiểu.

Có thể tạo ra các tiêu đề với mục đích duy nhất là bao gồm rất nhiều tiêu đề khác. Những điều này hiếm khi trở thành một ý tưởng hay về lâu dài vì một số tệp nguồn sẽ thực sự cần tất cả các phương tiện được mô tả bởi tất cả các tiêu đề. Ví dụ: a <standard-c.h>có thể được tạo ra bao gồm tất cả các tiêu đề C tiêu chuẩn - hãy cẩn thận vì một số tiêu đề không phải lúc nào cũng có mặt. Tuy nhiên, rất ít chương trình thực sự sử dụng các tiện ích của <locale.h> hoặc <tgmath.h>.

Xem thêm How to link multiple implementation files in C?

## NOTATION VÀ MISCELLANY

Tiêu chuẩn C nói rằng không có sự khác biệt lớn giữa ký hiệu **#include <header.h>** và **#include "header.h"**.

* [#include <header.h>] tìm kiếm một chuỗi các vị trí được xác định bởi cài đặt để tìm một tệp header được xác định duy nhất bởi chuỗi cụ thể giữa các dấu < và > và gây ra việc thay thế chỉ thị đó bằng toàn bộ nội dung của tệp header. Cách xác định các vị trí hoặc xác định tệp header là do cài đặt quy định.
* [#include "header.h"] gây ra việc thay thế chỉ thị đó bằng toàn bộ nội dung của tệp nguồn được xác định bởi chuỗi cụ thể giữa các dấu "…". Tệp nguồn có tên được tìm kiếm theo cách do cài đặt quy định. Nếu tìm kiếm này không được hỗ trợ hoặc nếu tìm kiếm thất bại, chỉ thị sẽ được xử lý lại như nếu nó đã đọc [#include <header.h>]...

Vì vậy, dạng bằng ngoặc kép có thể tìm kiếm nhiều nơi hơn so với dạng bằng ngoặc nhọn. Tiêu chuẩn chỉ ra ví dụ rằng các tệp tiêu chuẩn nên được bao gồm trong ngoặc nhọn, ngay cả khi việc biên dịch hoạt động nếu bạn sử dụng dấu ngoặc kép thay vì đó. Tương tự, các tiêu chuẩn như POSIX sử dụng định dạng bằng ngoặc nhọn - và bạn cũng nên vậy. Dành tiêu chuẩn bằng ngoặc kép cho các tiêu chuẩn được xác định bởi dự án. Đối với các tiêu chuẩn được xác định bên ngoài (bao gồm cả các tiêu chuẩn từ các dự án khác mà dự án của bạn phụ thuộc vào), định dạng bằng ngoặc nhọn là phù hợp nhất.

Chú ý rằng phải có một khoảng trống giữa #include và tiêu đề, ngay cả khi các trình biên dịch sẽ chấp nhận không có khoảng trống ở đó. Khoảng trống rất hiệu quả.

Nhiều dự án sử dụng biểu thức như:

|  |
| --- |
| #include <openssl/ssl.h>  #include <sys/stat.h>  #include <linux/kernel.h> |

Bạn nên cân nhắc xem có nên sử dụng điều khiển không gian tên đó trong dự án của mình hay không (có lẽ đây là một ý tưởng hay). Bạn nên tránh xa các tên được sử dụng bởi các dự án hiện có (đặc biệt, cả hai sysvà linuxsẽ là những lựa chọn tồi).

Nếu bạn sử dụng điều này, mã của bạn phải cẩn thận và nhất quán trong việc sử dụng ký hiệu.

Không sử dụng ký hiệu **#include "../include/header.h**".

Tệp header hiếm khi định nghĩa biến. Mặc dù bạn sẽ giữ các biến toàn cầu ở mức tối thiểu, nếu bạn cần một biến toàn cầu, bạn sẽ khai báo nó trong tệp header và định nghĩa nó trong một tệp nguồn phù hợp, và tệp nguồn đó sẽ bao gồm tệp header để kiểm tra lại khai báo và định nghĩa, và tất cả các tệp nguồn sử dụng biến đó sẽ sử dụng tệp header để khai báo nó.

Hậu quả: bạn sẽ không khai báo biến toàn cầu trong một tệp nguồn - tệp nguồn chỉ chứa định nghĩa.

Các tệp tiêu đề hiếm khi khai báo staticcác hàm, với ngoại lệ đáng chú ý là static inlinecác hàm sẽ được xác định trong các tiêu đề nếu chức năng đó cần thiết trong nhiều tệp nguồn.

* Các tệp nguồn xác định các biến toàn cục và các hàm toàn cục.
* Các tệp nguồn không khai báo sự tồn tại của các biến hoặc hàm toàn cục; chúng bao gồm tiêu đề khai báo biến hoặc hàm.
* Các tệp tiêu đề khai báo các hàm và biến toàn cục (và các loại và tài liệu hỗ trợ khác).
* Các tệp tiêu đề không xác định các biến hoặc bất kỳ hàm nào ngoại trừ các hàm (static) inline.

**Cross-references**

* Where to document functions in C?
* List of standard header files in C and C++
* Is inline without static or extern ever useful in C99?
* How do I use extern to share variables between source files?
* What are the benefits of a relative path such as "../include/header.h" for a header?
* Header inclusion optimization
* Should I include every header?

## IDEMPOTENCE

Nếu một header file cụ thể được included nhiều lần trong một đơn vị dịch thuật (TU), sẽ không có bất kỳ sự cố biên dịch nào. Điều này được gọi là ''idempotence'; headers của bạn phải idempotent. Hãy nghĩ xem cuộc sống sẽ khó khăn như thế nào nếu bạn phải đảm bảo điều đó #include <stdio.h>chỉ được included vào một lần.

Có hai cách để đạt được idempotence: **header guards** và chỉ thị **#pragma once**.

**Header guards**

Header guardsđơn giản, đáng tin cậy và phù hợp với tiêu chuẩn C. Các dòng không chú thích đầu tiên trong header file phải có dạng:

|  |
| --- |
| #ifndef UNIQUE\_ID\_FOR\_HEADER  #define UNIQUE\_ID\_FOR\_HEADER |

Dòng không chú thích cuối cùng phải là #endif, tùy chọn có chú thích sau đó:

|  |
| --- |
| #endif /\* UNIQUE\_ID\_FOR\_HEADER \*/ |

Tất cả mã hoạt động, bao gồm các chỉ thị #include khác, nên nằm giữa những dòng này.

Mỗi tên phải là duy nhất. Thường, một hệ thống tên như HEADER\_H\_INCLUDED được sử dụng. Một số mã cũ hơn sử dụng một ký hiệu được xác định là Header guards(ví dụ: #ifndef BUFSIZ trong <stdio.h>), nhưng nó không đáng tin cậy như một tên duy nhất. Một tùy chọn có thể là sử dụng tên bảo vệ tiêu chuẩn được tạo ra từ mã MD5 (hoặc mã khác). Bạn nên tránh mô phỏng các kế hoạch được sử dụng bởi các tiêu chuẩn hệ thống mà thường sử dụng các tên đã được dành cho cài đặt - các tên bắt đầu bằng một gạch dưới theo sau bởi một gạch dưới khác hoặc một chữ cái viết hoa.

**The #pragma once Directive**

Một cách khác, một số trình biên dịch hỗ trợ chỉ thị #pragma once có tác dụng tương tự như ba dòng được hiển thị cho header guards.

|  |
| --- |
| #pragma *once* |

Các trình biên dịch hỗ trợ #pragma once bao gồm MS Visual Studio và GCC và Clang. Tuy nhiên, nếu tính di động là một vấn đề, nên sử dụng cơ chế bảo vệ tiêu chuẩn hoặc sử dụng cả hai. Trình biên dịch hiện đại (các trình biên dịch hỗ trợ C89 hoặc mới hơn) được yêu cầu bỏ qua, mà không có bình luận, các chỉ thị pragma mà họ không nhận ra ('Mọi chỉ thị pragma như vậy không được nhận dạng bởi cài đặt sẽ bị bỏ qua'), nhưng các phiên bản cũ của GCC không hoàn toàn như vậy.

## INCLUDE WHAT YOU USE (IWYU)

Dự án Include What You Use của Google, hoặc IWYU, đảm bảo rằng các tệp nguồn bao gồm tất cả các tiêu chuẩn được sử dụng trong mã code.

Giả sử một tệp nguồn source.c bao gồm một tiêu chuẩn arbitrary.h, tiêu chuẩn này trong lúc đó tình cờ bao gồm tiêu chuẩn freeloader.h, nhưng tệp nguồn cũng một cách rõ ràng và độc lập sử dụng các chức năng từ freeloader.h. Mọi thứ bắt đầu tốt đẹp.

Sau đó, một ngày nào đó, arbitrary.h được thay đổi để người dùng của nó không còn cần các chức năng của freeloader.h nữa. Đột nhiên, source.c ngừng biên dịch - vì nó không đáp ứng được tiêu chí IWYU. Vì mã code trong source.c một cách rõ ràng sử dụng các chức năng của freeloader.h, nó nên đã bao gồm những gì nó sử dụng - nên có một chỉ thị #include "freeloader.h" một cách rõ ràng trong nguồn nguồn. (Tính đơn vị đã đảm bảo không có vấn đề nào.)

Triết lý của IWYU tối đa hóa khả năng rằng mã code tiếp tục biên dịch ngay cả khi có những thay đổi hợp lý được thực hiện đối với các giao diện. Rõ ràng, nếu mã code của bạn gọi một chức năng sau đó bị loại bỏ khỏi giao diện đã công bố, không có lượng chuẩn bị nào có thể ngăn ngừa sự cần thiết của các thay đổi. Đây là lý do tại sao các thay đổi đối với APIs được tránh nếu có thể, và tại sao có chu kỳ bỏ phiên bản qua nhiều phiên bản phát hành, v.v.

Điều này là một vấn đề đặc biệt trong C++ vì các tiêu chuẩn tiêu chuẩn được phép bao gồm lẫn nhau. Tệp nguồn file.cpp có thể bao gồm một tiêu chuẩn header1.h mà trên một nền tảng nào đó bao gồm một tiêu chuẩn tiêu chuẩn khác header2.h. file.cpp có thể sử dụng cả chức năng của header2.h nữa. Ban đầu điều này không phải là một vấn đề - mã code sẽ được biên dịch vì header1.h bao gồm header2.h. Trên nền tảng khác hoặc khi nâng cấp nền tảng hiện tại, header1.h có thể được điều chỉnh để không còn bao gồm header2.h nữa, và sau đó file.cpp sẽ ngừng biên dịch như kết quả. IWYU sẽ phát hiện vấn đề và đề xuất rằng header2.h nên được bao gồm trực tiếp trong file.cpp. Điều này sẽ đảm bảo nó tiếp tục biên dịch. Các yếu tố tương tự cũng áp dụng cho mã code C.

# <CTYPE.H> - CHARACTER CLASSFICATION & CONVERTER

## GIỚI THIỆU

Tiêu chuẩn ctype.h là một phần của thư viện chuẩn C. Nó cung cấp các chức năng để phân loại và chuyển đổi các ký tự.

Tất cả các hàm này đều nhận một tham số, một int mà phải là EOF hoặc có thể biểu diễn như một unsigned char.

Tên của các hàm phân loại có tiền tố là 'is'. Mỗi giá trị trả về một giá trị số nguyên khác không (TRUE) nếu ký tự được truyền cho nó thỏa mãn điều kiện liên quan. Nếu điều kiện không được thỏa mãn thì hàm trả về giá trị 0 (FALSE).

Các hàm phân loại này hoạt động như được hiển thị, giả sử ngôn ngữ C mặc định:

|  |
| --- |
| int a;  int c = 'A';  a = isalpha(c); /\* Kiểm tra xem c có phải là chữ cái (A-Z, a-z), trả về giá trị khác không ở đây. \*/  a = isalnum(c); /\* Kiểm tra xem c có phải là chữ cái hoặc chữ số (A-Z, a-z, 0-9), trả về giá trị khác không ở đây. \*/  a = iscntrl(c); /\* Kiểm tra xem c có phải là ký tự điều khiển (0x00-0x1F, 0x7F), trả về giá trị zero ở đây. \*/  a = isdigit(c); /\* Kiểm tra xem c có phải là số (0-9), trả về giá trị zero ở đây. \*/  a = isgraph(c); /\* Kiểm tra xem c có biểu diễn đồ họa (bất kỳ ký tự in nào trừ dấu cách),trả về giá trị khác không ở đây. \*/  a = islower(c); /\* Kiểm tra xem c có phải là chữ thường (a-z), trả về giá trị zero ở đây. \*/  a = isprint(c); /\* Kiểm tra xem c có phải là bất kỳ ký tự có thể in (bao gồm cả dấu cách), trả về giá trị khác không ở đây. \*/  a = isupper(c); /\* Kiểm tra xem c có phải là chữ in hoa (a-z), trả về giá trị zero ở đây. \*/  a = ispunct(c); /\* Kiểm tra xem c có phải là ký tự dấu câu, trả về giá trị zero ở đây. \*/  a = isspace(c); /\* Kiểm tra xem c có phải là ký tự khoảng trắng, trả về giá trị zero ở đây. \*/  a = isupper(c); /\* Kiểm tra xem c có phải là chữ in hoa (A-Z), trả về giá trị khác không ở đây. \*/  a = isxdigit(c); /\* Kiểm tra xem c có phải là chữ số thập lục phân (A-F, a-f, 0-9), trả về giá trị khác không ở đây. \*/ |

Phiên bản ≥ C99

|  |
| --- |
| a = isblank(c); /\* Kiểm tra xem c có phải là ký tự trống (khoảng trắng hoặc tab), trả về giá trị khác không ở đây. \*/ |

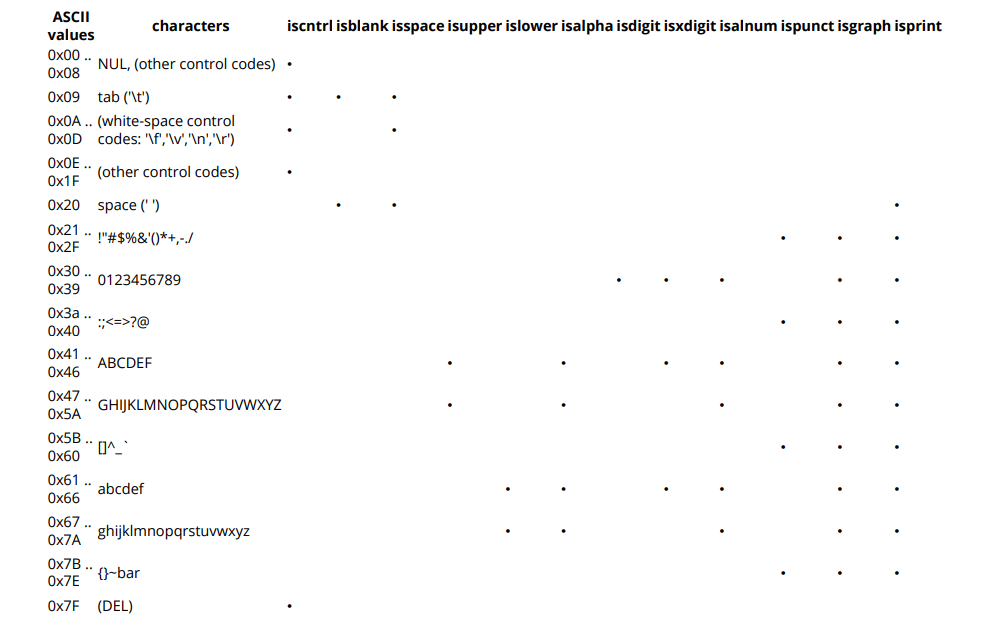
Có hai hàm chuyển đổi. Chúng được đặt tên bằng tiền tố 'to'. Các hàm này nhận cùng đối số như các hàm ở trên. Tuy nhiên, giá trị trả về không phải là giá trị zero hoặc không phải giá trị không mà là đối số được truyền vào đã được thay đổi theo một cách nào đó.

Các hàm chuyển đổi này hoạt động như sau, giả sử với ngữ cảnh C locale mặc định:

|  |
| --- |
| int a;  int c = 'A';  /\* Chuyển đổi c thành chữ thường (a-z).  \* Nếu không thể chuyển đổi, giá trị không thay đổi được trả về.  \* Trả về 'a' ở đây.  \*/  a = tolower(c);  /\* Chuyển đổi c thành chữ in hoa (A-Z).  \* Nếu không thể chuyển đổi, giá trị không thay đổi được trả về.  \* Trả về 'A' ở đây.  \*/  a = toupper(c); |

Thông tin dưới đây được trích từ trang web cplusplus.com, mô tả cách 127 ký tự trong bộ mã ASCII ban đầu được xem xét bởi mỗi hàm phân loại (một dấu • chỉ ra rằng hàm trả về khác không cho ký tự đó):

Giá trị ASCII của các ký tự



## PHÂN LOẠI CÁC KÍ TỰ ĐỌC TỪ MỘT LUỒNG

|  |
| --- |
| #include <ctype.h>  #include <stdio.h>  typedef struct {  *size\_t* space;  *size\_t* alnum;  *size\_t* punct;  } *chartypes*;  *chartypes* classify(*FILE* \**f*) {  *chartypes* types = { 0, 0, 0 };  int ch;    while ((ch = fgetc(*f*)) != EOF) {  types.space += !!isspace(ch);  types.alnum += !!isalnum(ch);  types.punct += !!ispunct(ch);  }    return types;  } |

Hàm classify đọc các ký tự từ một luồng (stream) và đếm số lượng dấu cách, ký tự chữ và ký tự dấu câu. Nó tránh một số lỗi phổ biến.

* Khi đọc một ký tự từ luồng, kết quả được lưu dưới dạng int, vì không thì sẽ có một sự mơ hồ giữa việc đọc EOF (ký hiệu cuối tập tin) và một ký tự có cùng mẫu bit.
* Các hàm phân loại ký tự (ví dụ isspace) mong đợi đối số của chúng có thể biểu diễn dưới dạng unsigned char hoặc giá trị của macro EOF. Vì vậy, điều này chính xác là điều mà fgetc trả về, không cần phải chuyển đổi ở đây.
* Giá trị trả về của các hàm phân loại ký tự chỉ phân biệt giữa zero (nghĩa là sai) và không phải zero (nghĩa là đúng). Đối với việc đếm số lần xuất hiện, giá trị này cần được chuyển đổi thành 1 hoặc 0, điều này được thực hiện bằng cách phủ định kép, !!.

## PHÂN LOẠI CÁC KÍ TỰ ĐỌC TỪ MỘT CHUỖI

|  |
| --- |
| #include <ctype.h>  #include <stddef.h>  typedef struct {  *size\_t* space;  *size\_t* alnum;  *size\_t* punct;  } *chartypes*;  *chartypes* classify(const char \**s*) {  *chartypes* types = { 0, 0, 0 };  const char \*p;  for (p = *s*; \*p != '\0'; p++) {  types.space += !!isspace((unsigned char)\*p);  types.alnum += !!isalnum((unsigned char)\*p);  types.punct += !!ispunct((unsigned char)\*p);  }  return types;  } |

Hàm classify kiểm tra tất cả các ký tự từ một chuỗi và đếm số lượng ký tự dấu cách, ký tự chữ và ký tự dấu câu. Nó tránh một số rủi ro.

* Các hàm phân loại ký tự (ví dụ isspace) mong đợi đối số của chúng có thể được biểu diễn dưới dạng unsigned char hoặc giá trị của macro EOF.
* Biểu thức \*p có kiểu char và do đó phải được chuyển đổi để phù hợp với văn bản ở trên.
* Kiểu char được xác định tương đương với hoặc signed char hoặc unsigned char.
* Khi char tương đương với unsigned char, không có vấn đề gì, vì mọi giá trị có thể có của kiểu char có thể được biểu diễn dưới dạng unsigned char.
* Khi char tương đương với signed char, nó phải được chuyển đổi thành unsigned char trước khi được truyền vào các hàm phân loại ký tự. Và mặc dù giá trị của ký tự có thể thay đổi do việc chuyển đổi này, đây chính xác là điều các hàm này mong đợi.
* Giá trị trả về của các hàm phân loại ký tự chỉ phân biệt giữa số không (nghĩa là sai) và số khác không (nghĩa là đúng). Để đếm số lần xuất hiện, giá trị này cần được chuyển đổi thành 1 hoặc 0, điều này được thực hiện bằng cách phủ định kép, !!.

# SIDE EFFECTS

## TOÁN TỬ TĂNG/GIẢM TRƯỚC/SAU

Trong C, có hai toán tử một ngôi - '++' và '--' - là nguồn gây nhầm lẫn rất phổ biến. Toán tử ++ được gọi là toán tử tăng và toán tử -- được gọi là toán tử giảm. Cả hai đều có thể được sử dụng ở dạng **tiền tố** hoặc **hậu tố**. Cú pháp cho dạng tiền tố của toán tử ++ là **++toán\_hạng** và cú pháp cho dạng hậu tố là **toán\_hạng++**. Khi được sử dụng ở dạng tiền tố, toán\_hạng được tăng trước 1 đơn vị và giá trị kết quả của toán\_hạng được sử dụng trong việc đánh giá biểu thức. Xem xét ví dụ sau:

|  |
| --- |
| int n, x = 5;  n = ++x; /\* x được tăng lên 1 (x=6), và kết quả được gán cho n (n=6) \*/  /\* Đây là một dạng ngắn gọn cho hai câu lệnh: \*/  /\* x = x + 1; \*/  /\* n = x; \*/ |

Khi được sử dụng ở dạng hậu tố, giá trị hiện tại của toán hạng được sử dụng trong biểu thức và sau đó giá trị của toán hạng được tăng thêm 1. Hãy xem xét ví dụ sau:

|  |
| --- |
| int n, x = 5;  n = x++; /\* giá trị của x (5) được gán trước cho n (5), sau đó x được tăng lên 1; x (6) \*/  /\* Đây là một dạng ngắn gọn cho hai câu lệnh: \*/  /\* n = x; \*/  /\* x = x + 1; \*/ |

Cách làm việc của toán tử giảm -- có thể được hiểu tương tự.

Đoạn mã sau đây mô tả những gì mỗi toán tử làm:

|  |
| --- |
| int main()  {  int a, b, x = 42;  a = ++x; /\* a và x là 43 \*/  b = x++; /\* b là 43, x là 44 \*/  a = x--; /\* a là 44, x là 43 \*/  b = --x; /\* b và x là 42 \*/  return 0;  } |

Từ trên, rõ ràng rằng các toán tử hậu trả về giá trị hiện tại của một biến và sau đó thay đổi nó, nhưng các toán tử tiền trả về giá trị đã được sửa đổi của biến.

Trong tất cả các phiên bản của C, thứ tự đánh giá của các toán tử tiền và hậu không được xác định, do đó đoạn mã sau có thể trả về kết quả không mong đợi:

|  |
| --- |
| int main()  {  int a, x = 42;  a = x++ + x; /\* sai \*/  a = x + x; /\* đúng \*/  ++x;  int ar[10];  x = 0;  ar[x] = x++; /\* sai \*/  ar[x++] = x; /\* sai \*/  ar[x] = x; /\* đúng \*/  ++x;  return 0;  } |

Chú ý rằng cũng là thực hành tốt khi sử dụng toán tử tiền thay vì hậu khi được sử dụng một mình trong một câu lệnh. Xem xét mã trên cho điều này.

Lưu ý rằng khi một hàm được gọi, tất cả các hiệu ứng phụ lên đối số phải diễn ra trước khi hàm chạy.

|  |
| --- |
| int foo(int *x*)  {  return *x*;  }  int main()  {  int a = 42;  int b = foo(a++); /\* Điều này trả về 43, ngay cả khi có vẻ như nó nên trả về 42 \*/  return 0;  } |

# MULTI – CHARACTER CHARACTER SEQUENCE

## TRIGRAPHS

Các ký hiệu [ ] { } ^ \ | ~ # thường được sử dụng trong các chương trình C. Nhưng vào cuối thập kỷ 1980, có các bộ mã đang được sử dụng (ví dụ như các biến thể ISO 646 tại các quốc gia Scandinavia), trong đó các vị trí ký tự ASCII này được sử dụng cho các ký tự biến thể của ngôn ngữ quốc gia (ví dụ: £ thay cho # ở Vương quốc Anh; Æ Å æ å ø Ø thay cho { } { } | \ ở Đan Mạch; không có ~ trong EBCDIC). Điều này làm cho việc viết mã C trên các máy sử dụng các bộ mã này trở nên khó khăn.

Để giải quyết vấn đề này, chuẩn C đề xuất sử dụng các kết hợp ba ký tự để tạo ra một ký tự đơn gọi là trigraph. Một trigraph là một dãy gồm ba ký tự, trong đó hai ký tự đầu tiên là dấu hỏi ??. Mỗi trigraph tương ứng với một ký tự thay thế. Ví dụ, trigraph ??= tương ứng với ký tự #, trigraph ??< tương ứng với ký tự {.

Phần sau đây là một ví dụ đơn giản sử dụng trigraph thay vì #, { và }:

|  |
| --- |
| ??=include <stdio.h>  int main()  ??<  printf("Hello World!\n");  ??> |

Đoạn mã trên sẽ được xử lý bởi bộ tiền xử lý C bằng cách thay thế trigraph bằng các ký tự đơn tương ứng như khi mã đã được viết:

|  |
| --- |
| #include <stdio.h>  int main()  {  printf("Hello World!\n");  } |

|  |  |
| --- | --- |
| **Trigraph** | **Equivalent** |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

Lưu ý rằng trigraphs có thể gây rối vì, ví dụ, ??/ là một dấu gạch chéo ngược và có thể ảnh hưởng đến ý nghĩa của các dòng tiếp tục trong các chú thích, và chúng phải được nhận dạng trong chuỗi và ký tự literal (ví dụ: '??/??/' là một ký tự duy nhất, là một dấu gạch chéo ngược).

## DIGRAPHS

Phiên bản ≥ C99

Năm 1994, các thay thế dễ đọc hơn cho năm trigraph phổ biến đã được cung cấp. Chúng sử dụng chỉ hai ký tự và được gọi là digraph. Khác với trigraph, digraph là các token. Nếu một digraph xuất hiện trong một token khác (ví dụ: chuỗi hoặc hằng ký tự) thì nó sẽ không được xử lý như một digraph mà vẫn giữ nguyên như cũ.

Dưới đây là sự khác biệt trước và sau khi xử lý digraphs.

|  |
| --- |
| #include <stdio.h>  int main()  <%  printf("Hello *%*> World!\n"); /\* Note that the string contains a digraph \*/  %> |

Sẽ được xử lý tương tự như:

|  |
| --- |
| #include <stdio.h>  int main()  {  printf("Hello *%*> World!\n"); /\* Note the unchanged digraph within the string. \*/  } |

|  |  |
| --- | --- |
| **Digraph** | **Equivalent** |
| <: | [ |
| :> | ] |
| <% | { |
| %> | } |
| %: | # |

# RÀNG BUỘC (CONSTRAINT)

## TÊN BIẾN TRÙNG LẶP TRONG CÙNG PHẠM VI

Một ví dụ về ràng buộc được thể hiện trong tiêu chuẩn C là có hai biến cùng tên được khai báo trong cùng một phạm vi1), ví dụ:

|  |
| --- |
| void foo(int *bar*)  {  int var;  double var;  } |

Mã này vi phạm ràng buộc và phải tạo ra một thông báo chẩn đoán trong quá trình biên dịch. Điều này rất hữu ích so với hành vi không xác định vì nhà phát triển sẽ được thông báo về vấn đề trước khi chương trình được chạy, tránh nguy cơ xảy ra bất cứ điều gì.

Do đó, các ràng buộc thường là các lỗi dễ dàng phát hiện trong quá trình biên dịch như ví dụ trên, các vấn đề dẫn đến hành vi không xác định nhưng khó hoặc không thể phát hiện trong quá trình biên dịch không phải là các ràng buộc.

Đoạn văn chính xác:

Phiên bản = C99

Nếu một định danh không có liên kết, thì không nên có nhiều hơn một khai báo của định danh (trong một bộ khai báo hoặc chỉ mục kiểu) cùng một phạm vi và trong cùng không gian tên, trừ trường hợp của các thẻ như đã quy định trong 6.7.2.3.

## TOÁN TỬ SỐ HỌC MỘT NGÔI

Các toán tử số học một ngôi + và - chỉ có thể sử dụng trên các kiểu số học, do đó nếu ví dụ cố gắng sử dụng chúng trên một cấu trúc thì chương trình sẽ tạo ra một thông báo chẩn đoán, ví dụ:

|  |
| --- |
| struct *foo*  {  bool bar;  };  void baz(void)  {  struct *foo* testStruct;  -testStruct; /\* Điều này vi phạm ràng buộc nên phải tạo ra một thông báo chẩn đoán \*/  } |

# INLINING

## HÀM INLINING ĐƯỢC SỬ DỤNG TRONG HƠN MỘT TỆP NGUỒN

Đối với các hàm nhỏ mà thường được gọi thường xuyên, chi phí liên quan đến việc gọi hàm có thể chiếm một phần đáng kể của tổng thời gian thực thi của hàm đó. Một cách để cải thiện hiệu suất là loại bỏ chi phí gọi hàm.

Trong ví dụ này, chúng ta sử dụng bốn hàm (cộng với main()) trong ba tệp nguồn. Hai trong số đó (plusfive() và timestwo()) mỗi cái được gọi bởi hai cái khác trong "source1.c" và "source2.c". main() được bao gồm để chúng ta có một ví dụ hoạt động.

**main.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include "headerfile.h"  int main(void) {  int start = 3;  int intermediate = complicated1(start);  printf("First result is %d\n", intermediate);  intermediate = complicated2(start);  printf("Second result is %d\n", intermediate);  return 0;  } |

**source1.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include "headerfile.h"  int complicated1(int *input*) {  int tmp = timestwo(*input*);  tmp = plusfive(tmp);  return tmp;  } |

**source2.c:**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include "headerfile.h"  int complicated2(int *input*) {  int tmp = plusfive(*input*);  tmp = timestwo(tmp);  return tmp;  } |

**headerfile.h:**

|  |
| --- |
| #ifndef HEADERFILE\_H  #define HEADERFILE\_H  int complicated1(int *input*);  int complicated2(int *input*);  inline int timestwo(int *input*) {  return *input* \* 2;  }  inline int plusfive(int *input*) {  return *input* + 5;  }  #endif |

Hàm timestwo và plusfive được gọi bởi cả complicated1 và complicated2, chúng nằm trong các "đơn vị dịch" hoặc các tệp nguồn khác nhau. Để sử dụng chúng theo cách này, chúng ta phải định nghĩa chúng trong tệp header.

Biên dịch như sau, giả sử gcc:

|  |
| --- |
| cc -O2 -std=c99 -c -o main.o main.c  cc -O2 -std=c99 -c -o source1.o source1.c  cc -O2 -std=c99 -c -o source2.o source2.c  cc main.o source1.o source2.o -o main |

Chúng ta sử dụng tùy chọn tối ưu hóa -O2 vì một số trình biên dịch không tích hợp mà không có tùy chọn tối ưu hóa được bật.

Tác động của từ khóa inline là biểu tượng hàm cụ thể không được phát ra vào tệp đối tượng. Nếu không có inline, một lỗi sẽ xảy ra ở dòng cuối cùng, nơi chúng ta đang liên kết các tệp đối tượng để tạo thành chương trình thực thi cuối cùng. Nếu không có inline, biểu tượng cùng tên sẽ được định nghĩa trong cả hai tệp .o và lỗi "biểu tượng được định nghĩa nhiều lần" sẽ xảy ra.

Trong những tình huống mà biểu tượng thực sự cần thiết, có hai khả năng để xử lý vấn đề đó. Khả năng đầu tiên là thêm một khai báo extern bổ sung của các hàm được tích hợp trong chính xác một trong các tệp .c. Vì vậy, thêm dòng sau vào source1.c:

|  |
| --- |
| extern int timestwo(int *input*);  extern int plusfive(int *input*); |

Khả năng thứ hai là định nghĩa hàm với static inline thay vì inline. Phương pháp này có điểm bất lợi là cuối cùng một bản sao của hàm cụ thể có thể được tạo ra trong mọi tệp đối tượng được tạo ra bằng tệp header này.

# UNIONS

## SỬ DỤNG UNIONS ĐỂ TÁI TẠO CÁC GIÁ TRỊ

Một số phiên bản C cho phép mã viết vào một thành viên của kiểu Union sau đó đọc từ một thành viên khác để thực hiện một loại ép kiểu diễn giải lại (phân tích cú pháp kiểu mới như biểu diễn bit của kiểu cũ).

Tuy nhiên, điều quan trọng cần lưu ý là điều này không được phép theo tiêu chuẩn C hiện tại hoặc quá khứ và sẽ dẫn đến hành vi không xác định, không hơn không kém, đây là một tiện ích mở rộng rất phổ biến được cung cấp bởi trình biên dịch (vì vậy hãy kiểm tra tài liệu trình biên dịch của bạn nếu bạn định làm điều này) .

Một ví dụ thực tế về kỹ thuật này là thuật toán "Fast Inverse Square Root," mà dựa vào các chi tiết về số dấu phẩy động IEEE 754 để thực hiện tính căn bình phương nghịch đảo nhanh hơn so với việc sử dụng phép tính số dấu phẩy động. Thuật toán này có thể được thực hiện thông qua việc chuyển kiểu con trỏ (rất nguy hiểm và vi phạm quy tắc đa loại nghiêm ngặt) hoặc thông qua union (vẫn là hành vi không xác định nhưng hoạt động trên nhiều trình biên dịch):

|  |
| --- |
| union *floatToInt*  {  int32\_t intMember;  float floatMember; /\* Float phải là 32 bit IEEE 754 để tính năng này hoạt động \*/  };  float inverseSquareRoot(float *input*)  {  union *floatToInt* x;  int32\_t i;  float f;  x.floatMember = *input*; /\* Gán vào thành viên kiểu float \*/  i = x.intMember; /\* Đọc trở lại từ thành viên kiểu integer \*/  i = 0x5f3759df - (i >> 1);  x.intMember = i; /\* Gán vào thành viên kiểu integer \*/  f = x.floatMember; /\* Đọc trở lại từ thành viên kiểu float \*/  f = f \* (1.5f - *input* \* 0.5f \* f \* f);  return f \* (1.5f - *input* \* 0.5f \* f \* f);  } |

Kỹ thuật này đã được sử dụng rộng rãi trong đồ họa máy tính và trò chơi trước đây do tốc độ nhanh hơn so với sử dụng các phép toán dấu phẩy động và rất thỏa hiệp, mất đi một số độ chính xác và rất khó di chuyển để đổi lấy tốc độ.

## VIẾT VÀO MỘT PHẦN TỬ UNION VÀ ĐỌC TỪ PHẦN TỬ KHÁC

Các thành viên của một liên hợp chia sẻ cùng một vùng bộ nhớ. Điều này có nghĩa rằng việc ghi vào một thành viên sẽ ghi đè lên dữ liệu trong tất cả các thành viên khác và việc đọc từ một thành viên sẽ dẫn đến việc đọc cùng dữ liệu như việc đọc từ tất cả các thành viên khác. Tuy nhiên, vì các thành viên của liên hợp có thể có các loại và kích thước khác nhau, dữ liệu được đọc có thể được diễn giải khác nhau, xem thêm

|  |
| --- |
| http://stackoverflow.com/documentation/c/1119/structs-and-unions/9399/using-unions-to-reinterpret-values |

Ví dụ đơn giản dưới đây minh họa một union có hai phần tử, cả hai đều có cùng kiểu dữ liệu. Nó cho thấy việc ghi vào phần tử m\_1 dẫn đến giá trị được ghi được đọc từ phần tử m\_2 và việc ghi vào phần tử m\_2 dẫn đến việc giá trị được ghi được đọc từ phần tử m\_1.

|  |
| --- |
| #include <stdio.h>  union *my\_union* /\* Định nghĩa liên hợp \*/  {  int m\_1;  int m\_2;  };  int main (void)  {  union *my\_union* u; /\* Khai báo union \*/  u.m\_1 = 1; /\* Ghi vào m\_1 \*/  printf("u.m\_2: %i\n", u.m\_2); /\* Đọc từ m\_2 \*/  u.m\_2 = 2; /\* Ghi vào m\_2 \*/  printf("u.m\_1: %i\n", u.m\_1); /\* Đọc từ m\_1 \*/  return 0;  } |

## SỰ KHÁC BIỆT GIỮA STRUCT VÀ UNION

Đoạn mã sau minh họa rằng các phần tử của union chia sẻ bộ nhớ trong khi các phần tử của struct không chia sẻ bộ nhớ.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  union *My\_Union*  {  int variable\_1;  int variable\_2;  };  struct *My\_Struct*  {  int variable\_1;  int variable\_2;  };  int main (void)  {  union *My\_Union* u;  struct *My\_Struct* s;  u.variable\_1 = 1;  u.variable\_2 = 2;  s.variable\_1 = 1;  s.variable\_2 = 2;  printf ("u.variable\_1: %i\n", u.variable\_1);  printf ("u.variable\_2: %i\n", u.variable\_2);  printf ("s.variable\_1: %i\n", s.variable\_1);  printf ("s.variable\_2: %i\n", s.variable\_2);  printf ("sizeof (union My\_Union): %i\n", sizeof (union *My\_Union*));  printf ("sizeof (struct My\_Struct): %i\n", sizeof (struct *My\_Struct*));  return 0;  } |

# THREADS (NATIVE)

## Khởi tạo một thread

Trong hầu hết các trường hợp, tất cả dữ liệu mà nhiều luồng truy cập nên được khởi tạo trước khi các luồng được tạo ra. Điều này đảm bảo rằng tất cả các luồng bắt đầu với một trạng thái rõ ràng và không xảy ra tình huống cạnh tranh.

Nếu điều này không khả thi, ta có thể sử dụng once\_flag và call\_once.

|  |
| --- |
| #include <threads.h>  #include <stdlib.h>  double const\* Big = 0;  static once\_flag onceBig = ONCE\_INIT;  void destroyBig(void) {  free((void\*)Big);  }  void initBig(void) {  double\* b = malloc(largeNum);  if (!b) {  perror("allocation failed for Big");  exit(EXIT\_FAILURE);  }  initializeBigWithSophisticatedValues(largeNum, b);  Big = b;  atexit(destroyBig);  at\_quick\_exit(destroyBig);  }  int myThreadFunc(void\* *a*) {  call\_once(&onceBig, initBig);  ...  return 0;  } |

once\_flag được sử dụng để điều phối các luồng khác nhau có thể muốn khởi tạo cùng dữ liệu Big. Cuộc gọi đến call\_once đảm bảo rằng:

* initBig được gọi chính xác một lần.
* call\_once chặn cho đến khi cuộc gọi initBig đã được thực hiện, bởi cùng một luồng hoặc một luồng khác.

Ngoài việc cấp phát, một việc điển hình để thực hiện trong một hàm được gọi một lần như vậy là khởi tạo động của các cấu trúc dữ liệu kiểm soát luồng như mtx\_t hoặc cnd\_t không thể được khởi tạo tĩnh, bằng cách sử dụng mtx\_init hoặc cnd\_init tương ứng.

## Bắt đầu nhiều luồng

|  |
| --- |
| #include <stdio.h>  #include <threads.h>  #include <stdlib.h>  struct *my\_thread\_data* {  double factor;  };  int my\_thread\_func(void\* *a*) {  struct *my\_thread\_data*\* d = *a*;  // thực hiện điều gì đó với d  printf("Chúng tôi tìm thấy %g\n", d->factor);  // trả về mã thành công hoặc mã lỗi  return d->factor > 1.0;  }  int main(int *argc*, char\* *argv*[*argc*+1]) {  unsigned n = 4;  if (*argc* > 1) n = strtoull(*argv*[1], 0, 0);  // dự trữ không gian cho các đối số cho các luồng  struct *my\_thread\_data* D[n]; // không thể khởi tạo  for (unsigned i = 0; i < n; ++i) {  D[i] = (struct *my\_thread\_data*){ .factor = 0.5\*i, };  }  // dự trữ không gian cho ID của các luồng  *thrd\_t* id[4];  // khởi chạy các luồng  for (unsigned i = 0; i < n; ++i) {  thrd\_create(&id[i], my\_thread\_func, &D[i]);  }  // Chờ tất cả các luồng kết thúc, nhưng bỏ đi các giá trị trả về của chúng  for (unsigned i = 0; i < n; ++i) {  thrd\_join(id[i], 0);  }  return EXIT\_SUCCESS;  } |

# ĐA LUỒNG (MULTITHREADING)

**Đa luồng (Multithreading)** là một form chuyên dụng của đa nhiệm (multitasking) là tính năng cho phép máy tính của bạn chạy hai hoặc nhiều chương trình đồng thời. Nói chung, có hai kiểu đa nhiệm là: process-based (dựa trên tiến trình) và thread-based (dựa trên luồng).

Đa nhiệm dựa trên tiến trình xử lý việc thực thi đồng thời của các chương trình. Đa nhiệm dựa trên luồng xử lý việc thực thi đồng thời các phần của cùng một chương trình.

Một chương trình đa luồng chứa hai hoặc nhiều phần mà có thể chạy đồng thời. Mỗi phần của chương trình đó được gọi là một thread, và mỗi thread định nghĩa một path riêng biệt của sự thực thi.

C++ không chứa bất kỳ hỗ trợ có sẵn nào cho các ứng dụng đa luồng. Thay vào đó, nó dựa hoàn toàn vào Hệ điều hành để cung cấp tính năng này.

Chương này giả sử bạn đang làm việc trên Hệ điều hành Linux và chúng tôi đang chuẩn bị viết chương trình đa luồng trong C++ bởi sử dụng POSIX. POSIX Threads hoặc Pthreads cung cấp API mà có sẵn trên nhiều hệ thống như FreeBSD, NetBSD, GNU/Linux, Mac OS X và Solaris.

## Tạo Thread trong C++

Đây là chương trình chúng ta sử dụng để tạo một POSIX thread:

|  |
| --- |
| #**include** <pthread.h>  pthread\_create (thread, attr, start\_routine, arg) |

Ở đây, **pthread\_create** tạo một thread mới và làm nó có thể thực thi. Chương trình này có thể được gọi bất cứ thời điểm nào ở bất cứ đâu trong code của bạn. Dưới đây là miêu tả các tham số:

|  |  |
| --- | --- |
| **Tham số** | **Miêu tả** |
| thread | Một định danh duy nhất cho thread mới được trả về bởi chương trình con |
| attr | Một thuộc tính mà có thể được sử dụng để thiết lập các thuộc tính của thread. Bạn có thể xác định một đối tượng thuộc tính thread, hoặc NULL cho các giá trị mặc định |
| start\_routine | Chương trình C++ mà thread này sẽ thực thi một khi nó được tạo |
| arg | Một tham số đơn mà có thể được truyền tới start\_routine. Nó phải được truyền bởi tham chiếu dạng một con trỏ của kiểu void. NULL có thể được sử dụng nếu không có tham số nào được truyền |

Số thread tối đa có thể được tạo bởi một tiến trình là phụ thuộc vào trình triển khai (Implementation). Một khi được tạo, các thread là ngang hàng, và có thể tạo các thread khác. Không có sự phụ thuộc giữa các thread trong C++.

## Kết thúc Thread trong C++

Chương trình sau được sử dụng để kết thúc một POSIX thread trong C++:

#**include** <pthread.h>

pthread\_exit (status)

Ở đây **pthread\_exit** được sử dụng để kết thúc một thread. Chương trình pthread\_exit() được gọi sau khi một thread đã hoàn thành công việc của nó và không cần thiết phải tồn tại nữa.

Nếu main() kết thúc trước các thread nó đã tạo, và kết thúc chương trình pthread\_create(), thì các thread khác sẽ tiếp tục thực thi. Nếu không thì, chúng sẽ tự động được kết thúc khi main() hoàn thành.

**Ví dụ**

Ví dụ đơn giản sau tạo 5 thread với chương trình pthread\_create(). Mỗi thread in một thông báo "Hello World!", và sau đó kết thúc với một lời gọi tới pthread\_exit() trong C++:

#**include** <iostream>

#**include** <cstdlib>

#**include** <pthread.h>using namespace std;#**define** SO\_THREAD 4void \*InLoiChao(void \*threadid)

{

cout << "Hello World! Thread ID la " << threadid << endl;

pthread\_exit(NULL);

}int main ()

{

pthread\_t threads[SO\_THREAD];

int rc;

int i;

for( i=0; i < SO\_THREAD; i++ ){

cout << "\nmain() : dang tao thread! " << i << endl;

rc = pthread\_create(&threads[i], NULL,

InLoiChao, (int \*)i);

if (rc){

cout << "\nError: Khong the tao thread!" << rc << endl;

exit(-1);

}

}

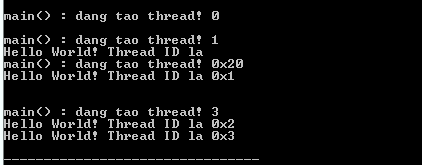
pthread\_exit(NULL);

}

Nếu bạn đang sử dụng command promt để biên dịch chương trình, thì bạn sử dụng thư viện **–lpthread** như sau:

$gcc test.cpp -lpthread

Nếu không, biên dịch và chạy chương trình C++ trên sẽ cho kết quả sau:



## Truyền tham số tới Thread trong C++

Ví dụ này minh họa cách truyền nhiều tham số thông qua một cấu trúc. Bạn có thể truyền bất kỳ kiểu dữ liệu nào trong một Thread callback, bởi vì nó trỏ tới void như được giải thích trong ví dụ sau:

#**include** <iostream>

#**include** <cstdlib>

#**include** <pthread.h>using namespace std;#**define** SO\_THREAD 5struct du\_lieu{

int id;

char \*thongdiep;

};void \*InLoiChao(void \*thamso)

{

struct du\_lieu \*data; data = (struct du\_lieu \*) thamso; cout << "\nThread ID la: " << data->id ;

cout << "\nThong diep: " << data->thongdiep << endl; pthread\_exit(NULL);

}int main ()

{

pthread\_t threads[SO\_THREAD];

struct du\_lieu td[SO\_THREAD];

int rc;

int i; for( i=0; i < SO\_THREAD; i++ ){

cout <<"\nmain() : dang tao thread! " << i << endl;

td[i].id = i;

td[i].thongdiep = "\nDay la thong diep";

rc = pthread\_create(&threads[i], NULL,

InLoiChao, (void \*)&td[i]);

if (rc){

cout << "\nError: Khong the tao thread! " << rc << endl;

exit(-1);

}

}

pthread\_exit(NULL);

}

Biên dịch và chạy chương trình C++ trên sẽ cho kết quả sau:



## Kết hợp và Tháo gỡ các Thread trong C++

Hai cú pháp sau được sử dụng để kết hợp (joining) hoặc tháo gỡ (detaching) các Thread trong C++:

pthread\_join (threadid, status)

pthread\_detach (threadid)

Chương trình con pthread\_join() đóng khối thread đang gọi tới khi threadid kết thúc. Khi một thread được tạo, một trong các thuộc tính định nghĩa nó là joinable hoặc detached. Chỉ các thread được tạo với dạng joinable có thể được kết hợp. Nếu một thread được tạo với dạng detached, nó không bao giờ được kết hợp.

Ví dụ sau minh họa cách đợi cho các thread kết thúc bởi sử dụng chương trình kết hợp Pthread trong C++:

#**include** <iostream>

#**include** <cstdlib>

#**include** <pthread.h>

#**include** <unistd.h>using namespace std;#**define** SO\_THREAD 5void \*wait(void \*tid)

{

int i;

sleep(1);

cout << "\nDang tam dung mot Thread " << endl;

cout << "\nThread voi ID la : " << tid << " ... dang thoat! " << endl;

pthread\_exit(NULL);

}int main ()

{

int rc;

int i;

pthread\_t threads[SO\_THREAD];

pthread\_attr\_t attr;

void \*status; // khoi tao va thiet lap mot thread co the ket hop

pthread\_attr\_init(&attr);

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_JOINABLE); for( i=0; i < SO\_THREAD; i++ ){

cout << "\nmain() : dang tao thread! " << i << endl;

rc = pthread\_create(&threads[i], NULL, wait, (void \*)i );

if (rc){

cout << "\nError: Khong the tao thread! " << rc << endl;

exit(-1);

}

} // giai phong thuoc tinh va doi thread khac

pthread\_attr\_destroy(&attr);

for( i=0; i < SO\_THREAD; i++ ){

rc = pthread\_join(threads[i], &status);

if (rc){

cout << "\nError: khong the ket hop! " << rc << endl;

exit(-1);

}

cout << "\nMain: ket huc thread id :" << i ;

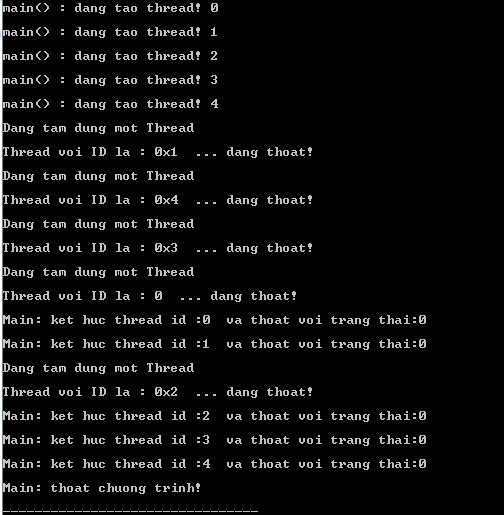
cout << " va thoat voi trang thai:" << status << endl;

} cout << "\nMain: thoat chuong trinh!" << endl;

pthread\_exit(NULL);

}

Biên dịch và chạy chương trình C++ trên sẽ cho kết quả sau:



Trong C11, có một thư viện chuẩn về Luồng Hoạt Động, <threads.h>, nhưng hiện chưa có trình biên dịch nào đã triển khai nó. Do đó, để sử dụng đa luồng trong C, bạn phải sử dụng các triển khai cụ thể cho từng nền tảng như thư viện luồng POSIX (thường được gọi là pthreads) bằng cách sử dụng tiêu đề pthread.h.

## VÍ DỤ ĐƠN GIẢN VỀ LUỒNG C11

|  |
| --- |
| #include <threads.h>  #include <stdio.h>  int run(void \**arg*)  {  printf("Hello world of C11 threads.");  return 0;  }  int main(int *argc*, const char \**argv*[])  {  *thrd\_t* thread;  int result;  thrd\_create(&thread, run, NULL);  thrd\_join(&thread, &result);    printf("Thread return %d at the end\n", result);  } |

# GIAO TIẾP GIỮA CÁC TIẾN TRÌNH (INTERPROCESS COMMUNICATION - IPC)

Cơ chế giao tiếp giữa các tiến trình (IPC) cho phép các tiến trình độc lập khác nhau tương tác với nhau. Ngôn ngữ C chuẩn không cung cấp bất kỳ cơ chế IPC nào. Do đó, tất cả các cơ chế như vậy được xác định bởi hệ điều hành mẹ. POSIX định nghĩa một tập hợp phức tạp các cơ chế IPC; Windows định nghĩa một tập hợp khác; và các hệ thống khác định nghĩa các biến thể riêng của chúng.

## Semaphores

Semaphore được sử dụng để đồng bộ hoạt động giữa hai hoặc nhiều tiến trình. POSIX định nghĩa hai tập hàm khác nhau cho semaphore:

* 'System V IPC' — semctl(), semop(), semget().
* 'POSIX Semaphores' — sem\_close(), sem\_destroy(), sem\_getvalue(), sem\_init(), sem\_open(), sem\_post(),sem\_trywait(), sem\_unlink().

Phần này mô tả semaphore theo kiểu System V IPC, được gọi là như vậy vì chúng bắt nguồn từ Unix System V.

Đầu tiên, bạn cần bao gồm các tiêu đề cần thiết. Phiên bản cũ của POSIX yêu cầu #include <sys/types.h>; nhưng POSIX hiện đại và hầu hết các hệ thống không yêu cầu điều này.

|  |
| --- |
| #include <sys/sem.h> |

Tiếp theo, bạn cần định nghĩa một khóa (key) trong cả tiến trình cha và con.

|  |
| --- |
| #define KEY 0x1111 |

Khóa này cần phải giống nhau trong cả hai chương trình hoặc chúng sẽ không tham chiếu đến cùng một cấu trúc IPC. Có cách để tạo ra một khóa được đồng ý mà không cần gắn cứng giá trị.

Sau đó, tùy theo trình biên dịch của bạn, bạn có thể cần làm bước này: khai báo một union cho mục đích thực hiện các thao tác semaphore.

|  |
| --- |
| union *semun* {  int val;  struct *semid\_ds* \*buf;  unsigned short \*array;  }; |

Sau đó, định nghĩa cấu trúc của bạn cho thử (semwait) và nâng (semsignal). Tên P và V bắt nguồn từ tiếng Hà Lan.

|  |
| --- |
| struct *sembuf* p = { 0, -1, SEM\_UNDO}; # semwait  struct sembuf v = { 0, +1, SEM\_UNDO}; # semsignal |

Bây giờ, bắt đầu bằng cách lấy ID cho semaphore IPC của bạn.

|  |
| --- |
| int id;  // Tham số thứ hai là số lượng semaphore  // Tham số thứ ba là chế độ (IPC\_CREAT tạo bộ semaphore nếu cần)  if ((id = semget(KEY, 1, 0666 | IPC\_CREAT)) < 0) {  /\* Xử lý lỗi \*/  } |

Trong tiến trình cha, khởi tạo semaphore để có một bộ đếm bằng 1.

|  |
| --- |
| union semun u;  u.val = 1;  if (semctl(id, 0, SETVAL, u) < 0) { // SETVAL là một macro chỉ định bạn đang đặt giá trị của semaphore  vào giá trị được xác định bởi union u  /\* Xử lý lỗi \*/  } |

Sau đó, bạn có thể giảm hoặc tăng semaphore theo nhu cầu. Ở đầu phần quan trọng của bạn, bạn giảm bộ đếm sử dụng hàm semop():

|  |
| --- |
| if (semop(id, &*p*, 1) < 0) {  /\* Xử lý lỗi \*/  } |

Để tăng semaphore, bạn sử dụng &v thay vì &p:

|  |
| --- |
| if (semop(id, &*v*, 1) < 0) {  /\* Xử lý lỗi \*/  } |

Lưu ý rằng mọi hàm trả về 0 khi thành công và -1 khi xảy ra lỗi. Không kiểm tra các trạng thái trả về này có thể gây ra các vấn đề nghiêm trọng.

**Ví dụ 1.1: Racing with Threads**

Chương trình dưới đây sẽ cho một tiến trình tạo một tiến trình con và cả hai tiến trình cha và con đều cố gắng in các ký tự lên terminal mà không có bất kỳ đồng bộ hóa nào.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  int main()  {  int pid;  pid = fork();  srand(pid);  if(pid < 0)  {  perror("fork"); exit(1);  }  else if(pid)  {  char \*s = "abcdefgh";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  }  }  else  {  char \*s = "ABCDEFGH";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  }  }  } |

Kết quả (lần chạy đầu tiên):

|  |
| --- |
| aAABaBCbCbDDcEEcddeFFGGHHeffgghh |

(Lần chạy thứ hai):

|  |
| --- |
| aabbccAABddBCeeCffgDDghEEhFFGGHH |

Biên dịch và chạy chương trình này sẽ cho bạn kết quả khác nhau mỗi lần chạy.

**Ví dụ 1.2:** **Avoid Racing with Semaphores**

Sửa đổi Ví dụ 1.1 để sử dụng semaphore, chúng ta có:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <string.h>  #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h>  #define KEY 0x1111  union semun {  int val;  struct semid\_ds \*buf;  unsigned short \*array;  };  struct *sembuf* p = { 0, -1, SEM\_UNDO};  struct *sembuf* v = { 0, +1, SEM\_UNDO};  int main()  {  int id = semget(KEY, 1, 0666 | IPC\_CREAT);  if(id < 0)  {  perror("semget"); exit(11);  }  union *semun* u;  u.val = 1;  if(semctl(id, 0, SETVAL, u) < 0)  {  perror("semctl"); exit(12);  }  int pid;  pid = fork();  srand(pid);  if(pid < 0)  {  perror("fork"); exit(1);  }  else if(pid)  {  char \*s = "abcdefgh";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  if(semop(id, &p, 1) < 0)  {  perror("semop p"); exit(13);  }  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  if(semop(id, &v, 1) < 0)  {  perror("semop p"); exit(14);  }  sleep(rand() % 2);  }  }  else  {  char \*s = "ABCDEFGH";  int l = strlen(s);  for(int i = 0; i < l; ++i)  {  if(semop(id, &p, 1) < 0)  {  perror("semop p"); exit(15);  }  putchar(s[i]);  fflush(stdout);  sleep(rand() % 2);  putchar(s[i]);  fflush(stdout);  if(semop(id, &v, 1) < 0)  {  perror("semop p"); exit(16);  }  sleep(rand() % 2);  }  }  } |

Kết quả

|  |
| --- |
| aabbAABBCCccddeeDDffEEFFGGHHgghh |

Biên dịch và chạy chương trình này sẽ cho bạn cùng một kết quả mỗi lần chạy.

# TESTING FRAMEWORKS

Nhiều nhà phát triển sử dụng các bài kiểm tra đơn vị (Unit Test) để kiểm tra xem phần mềm của họ hoạt động như mong đợi hay không. Các bài kiểm tra đơn vị (Unit Test) kiểm tra các đơn vị nhỏ của các phần mềm và đảm bảo rằng các đầu ra khớp với kỳ vọng.

Các framework giúp việc kiểm tra đơn vị dễ dàng hơn bằng cách cung cấp dịch vụ thiết lập/loại bỏ và phối hợp các bài kiểm tra.

Có nhiều framework kiểm thử đơn vị có sẵn cho C. Ví dụ, Unity là một framework hoàn toàn bằng ngôn ngữ C. Người ta thường sử dụng các framework kiểm thử C++ để kiểm thử mã C; cũng có nhiều framework kiểm thử C++.

## UNITY TEST FRAMEWORK

Unity là một framework kiểm thử theo kiểu xUnit dành cho việc kiểm thử đơn vị ứng dụng C. Nó được viết hoàn toàn bằng ngôn ngữ C và có khả năng chuyển giao, nhanh chóng, đơn giản, linh hoạt và có khả năng mở rộng. Nó được thiết kế đặc biệt để hữu ích cho việc kiểm thử đơn vị trong các hệ thống nhúng.

Một bài kiểm tra đơn giản kiểm tra giá trị trả về của một hàm có thể trông như sau:

|  |
| --- |
| void test\_FunctionUnderTest\_should\_ReturnFive(void)  {  TEST\_ASSERT\_EQUAL\_INT( 5, FunctionUnderTest() );  } |

Một tập tin kiểm thử đầy đủ có thể trông như sau:

|  |
| --- |
| #include "unity.h"  #include "UnitUnderTest.h" /\* Các đơn vị cần được kiểm thử. \*/  /\* Được chạy trước mỗi bài kiểm tra, đặt cuộc gọi khởi tạo đơn vị ở đây. \*/  void setUp (void) {}  /\* Được chạy sau mỗi bài kiểm tra, đặt cuộc gọi dọn dẹp đơn vị ở đây. \*/  void tearDown (void) {}  void test\_TheFirst(void) {  /\* Bỏ qua bài kiểm tra này nhưng in một thông báo. \*/  TEST\_IGNORE\_MESSAGE("Hello world!");  }  int main (void) {  UNITY\_BEGIN();  RUN\_TEST(test\_TheFirst); /\* Chạy bài kiểm tra. \*/  return UNITY\_END();  } |

Unity đi kèm với một số dự án mẫu, các tập tin makefile và một số kịch bản rake Ruby giúp việc tạo các tập tin kiểm tra dài hơn trở nên dễ dàng hơn một chút.

## Cmocka

CMocka là một framework kiểm thử đơn vị tinh tế dành cho ngôn ngữ C với hỗ trợ cho các đối tượng giả lập (mock objects). Nó chỉ yêu cầu thư viện tiêu chuẩn của C, hoạt động trên nhiều nền tảng tính toán khác nhau (bao gồm cả nhúng) và với các trình biên dịch khác nhau. Nó đi kèm với hướng dẫn kiểm thử với các đối tượng giả lập, tài liệu API và nhiều ví dụ khác nhau.

|  |
| --- |
| #include <stdarg.h>  #include <stddef.h>  #include <setjmp.h>  #include <cmocka.h>  void null\_test\_success (void \*\* *state*) {}  void null\_test\_fail (void \*\* *state*) {  assert\_true (0);  }  /\* Những hàm này sẽ được sử dụng để khởi tạo  và dọn dẹp tài nguyên sau mỗi lần chạy kiểm thử \*/  int setup (void \*\* *state*) {  return 0;  }  int teardown (void \*\* *state*) {  return 0;  }  int main (void) {  const struct *CMUnitTest* tests [] =  {  cmocka\_unit\_test (null\_test\_success),  cmocka\_unit\_test (null\_test\_fail),  };  /\* Nếu không cần đến các hàm setup và teardown,  ta có thể truyền NULL thay vào \*/  int count\_fail\_tests =  cmocka\_run\_group\_tests (tests, setup, teardown);    return count\_fail\_tests;  } |

Hãy nhớ rằng, CMocka là một framework kiểm thử đơn vị tinh tế dành cho C, và bạn có thể sử dụng nó để kiểm thử các đơn vị của mã C một cách hiệu quả và dễ dàng.

## Cpputests

CppUTest là một framework kiểm thử đơn vị theo kiểu xUnit dành cho C và C++. Nó được viết bằng C++ và mục tiêu là tính di động và đơn giản trong thiết kế. Nó hỗ trợ phát hiện rò rỉ bộ nhớ, tạo đối tượng giả lập và chạy các kiểm thử cùng với Google Test. CppUTest đi kèm với các tập lệnh hỗ trợ và các dự án mẫu cho Visual Studio và Eclipse CDT.

|  |
| --- |
| #include <CppUTest/CommandLineTestRunner.h>  #include <CppUTest/TestHarness.h>  TEST\_GROUP(*Foo\_Group*) {}  TEST(*Foo\_Group*, *Foo\_TestOne*) {}  /\* Bộ kiểm tra có thể được cung cấp các tùy chọn,  như kích hoạt đầu ra màu sắc, chạy chỉ một kiểm thử  cụ thể hoặc một nhóm kiểm thử, v.v. Điều này sẽ trả  về số lượng kiểm thử thất bại. \*/  int main(int *argc*, char \*\* *argv*) {  RUN\_ALL\_TESTS(*argc*, *argv*);  } |

Một nhóm kiểm thử có thể có các phương thức setup() và teardown(). Phương thức setup() được gọi trước mỗi kiểm thử và phương thức teardown() được gọi sau mỗi kiểm thử. Cả hai đều tùy chọn và có thể bị bỏ qua độc lập. Các phương thức và biến khác cũng có thể được khai báo trong một nhóm và sẽ có sẵn cho tất cả các kiểm thử của nhóm đó.

|  |
| --- |
| TEST\_GROUP(*Foo\_Group*) {  *size\_t* data\_bytes = 128;  void \* data;  void setup() {  data = malloc(data\_bytes);  }  void teardown() {  free(data);  }    void clear() {  memset(data, 0, data\_bytes);  }  } |

# VALGRIND

## MẤT DỮ LIỆU BYTE – QUÊN GIẢI PHÓNG BỘ NHỚ

Dưới đây là một chương trình sử dụng malloc nhưng không sử dụng free:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  int main(int *argc*, char \*\**argv*)  {  char \*s;  s = malloc(26); // nguyên nhân gây ra lỗi  return 0;  } |

Nếu không có đối số bổ sung, valgrind sẽ không tìm kiếm lỗi này.

Nhưng nếu chúng ta bật ***--leak-check=yes*** hoặc ***--tool=memcheck***, valgrind sẽ cảnh báo và hiển thị những dòng chịu trách nhiệm cho memory leaks nếu chương trình được biên dịch ở chế độ gỡ lỗi:

|  |
| --- |
| $ valgrind -q --leak-check=yes ./missing\_free  ==4776== 26 bytes in 1 blocks are definitely lost in loss record 1 of 1  ==4776== at 0x4024F20: malloc (vg\_replace\_malloc.c:236)  ==4776== by 0x80483F8: main (missing\_free.c:9)  ==4776== |

Nếu chương trình không được biên dịch ở chế độ gỡ lỗi (ví dụ: không có cờ -g trong GCC), valgrind vẫn sẽ chỉ ra chỗ xảy ra lỗi dựa trên hàm liên quan, nhưng không hiển thị dòng lỗi cụ thể.

Điều này giúp chúng ta quay trở lại và xem xét khối dữ liệu được cấp phát trong dòng đó và cố gắng theo dõi để xem tại sao nó không được giải phóng.

## CÁC LỖI PHỔ BIẾN THƯỜNG GẶP KHI SỬ DỤNG VALGRIND

Valgrind cung cấp cho bạn các dòng mà lỗi xảy ra ở cuối mỗi dòng trong định dạng (file.c:line\_no). Các lỗi trong valgrind được tóm tắt như sau:

|  |
| --- |
| ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0) |

Các lỗi phổ biến bao gồm:

1. **Lỗi đọc/ghi không hợp lệ**

|  |
| --- |
| ==8451== Invalid read of size 2  ==8451== at 0x4E7381D: getenv (getenv.c:84)  ==8451== by 0x4EB1559: \_\_libc\_message (libc\_fatal.c:80)  ==8451== by 0x4F5256B: \_\_fortify\_fail (fortify\_fail.c:37)  ==8451== by 0x4F5250F: \_\_stack\_chk\_fail (stack\_chk\_fail.c:28)  ==8451== by 0x40059C: main (valg.c:10)  ==8451== Address 0x700000007 is not stack'd, malloc'd or (recently) free'd |

Điều này xảy ra khi mã bắt đầu truy cập vào bộ nhớ không thuộc chương trình. Kích thước của bộ nhớ được truy cập cũng cho bạn một gợi ý về biến nào đã được sử dụng.

1. **Sử dụng biến chưa khởi tạo**

|  |
| --- |
| ==8795== 1 errors in context 5 of 8:  ==8795== Conditional jump or move depends on uninitialised value(s)  ==8795== at 0x4E881AF: vfprintf (vfprintf.c:1631)  ==8795== by 0x4E8F898: printf (printf.c:33)  ==8795== by 0x400548: main (valg.c:7) |

Theo lỗi, ở dòng 7 của hàm main của valg.c, cuộc gọi printf() truyền một biến chưa khởi tạo cho printf.

1. **Giải phóng bộ nhớ không hợp lệ**

|  |
| --- |
| ==8954== Invalid free() / delete / delete[] / realloc()  ==8954== at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==8954== by 0x4005A8: main (valg.c:10)  ==8954== Address 0x5203040 is 0 bytes inside a block of size 240 free'd  ==8954== at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==8954== by 0x40059C: main (valg.c:9)  ==8954== Block was alloc'd at  ==8954== at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload\_memcheck-amd64-linux.so)  ==8954== by 0x40058C: main (valg.c:7) |

Theo valgrind, mã giải phóng bộ nhớ một cách không hợp lệ (lần thứ hai) ở dòng 10 của valg.c, trong khi nó đã được giải phóng ở dòng 9 và khối bộ nhớ đã được cấp phát ở dòng 7.

## RUNNING VALGRIND

|  |
| --- |
| valgrind ./my-program arg1 arg2 < test-input |

Điều này sẽ chạy chương trình của bạn và tạo một báo cáo về bất kỳ phân bổ và giải phóng nào nó thực hiện. Nó cũng sẽ cảnh báo bạn về các lỗi thông thường như sử dụng bộ nhớ chưa khởi tạo, giải tham chiếu đến nơi lạ lẫm, viết ngoài phạm vi cuối khối được cấp phát bằng malloc, hoặc không giải phóng khối.

## ADDING FLAGS

Bạn cũng có thể bật thêm các kiểm tra, ví dụ:

|  |
| --- |
| valgrind -q --tool=memcheck --leak-check=yes ./my-program arg1 arg2 < test-input |

Xem ***valgrind --help*** để biết thêm thông tin về các tùy chọn (rất nhiều), hoặc xem tài liệu tại ***http://valgrind.org/*** để biết thông tin chi tiết về ý nghĩa của đầu ra.

# CÁC BIỂU THỨC VÀ THỰC TIỄN PHÁT TRIỂN PHỔ BIẾN TRONG LẬP TRÌNH C

## So sánh giá trị chữ và biến

Giả sử bạn đang so sánh một giá trị với một biến:

|  |
| --- |
| if (i == 2) // Cách không tốt  {  doSomething;  } |

Bây giờ giả sử bạn đã nhầm lẫn giữa == và =. Sau đó, bạn sẽ mất thời gian để nhận ra điều đó.

|  |
| --- |
| if (2 == i) // Cách tốt  {  doSomething;  } |

Khi đó, nếu dấu bằng được để thiếu, trình biên dịch sẽ cảnh báo về "cố gắng gán một giá trị chữ."

Điều này không bảo vệ bạn khi so sánh hai biến, nhưng mỗi điều nhỏ như vậy cũng giúp ích.

## Đừng để danh sách tham số của hàm trống – sử dụng “void”

Giả sử bạn đang tạo một hàm không yêu cầu đối số khi nó được gọi và bạn đang phải đối mặt với tình thế tiến thoái lưỡng nan về cách bạn nên xác định danh sách tham số trong nguyên mẫu hàm và định nghĩa hàm.

* Bạn có lựa chọn để giữ danh sách tham số trống cho cả nguyên mẫu và định nghĩa hàm. Do đó, chúng trông giống như câu lệnh gọi hàm mà bạn sẽ cần.
* Bạn đã đọc một nơi nào đó rằng một trong các ứng dụng của từ khóa "void" (chỉ có vài ứng dụng), là để định nghĩa danh sách tham số của các hàm không chấp nhận bất kỳ đối số nào trong lời gọi của chúng. Vì vậy, đây cũng là một lựa chọn.

Vậy, lựa chọn nào là đúng?

**Trả lời:** sử dụng từ khóa **"void"**

**Lời khuyên chung:** Nếu một ngôn ngữ cung cấp một số tính năng cụ thể để sử dụng cho một mục đích đặc biệt, bạn nên sử dụng nó trong mã của bạn. Ví dụ, sử dụng **enum** thay vì các macro #define (điều đó dành cho ví dụ khác).

Phần 6.7.6.3 "Khái báo hàm" của tiêu chuẩn C11, đoạn 10, nêu rõ:

Trường hợp đặc biệt của một tham số vô danh có kiểu void là mục duy nhất trong danh sách này cho biết rằng hàm không có tham số.

Đoạn 14 của cùng mục đó cũng chỉ ra sự khác biệt duy nhất:

... Một danh sách trống trong một khai báo hàm mà là một phần của định nghĩa của hàm đó chỉ ra rằng hàm không có tham số. Danh sách trống trong một khai báo hàm không phải là một phần của định nghĩa của hàm đó chỉ ra rằng không cung cấp thông tin về số hoặc kiểu tham số.

Một giải thích đơn giản được cung cấp bởi K&R (trang 72-73) cho các thông tin trên:

Hơn nữa, nếu một khai báo hàm không bao gồm các tham số, như double atof();, đó cũng được coi là không có gì được giả định về các đối số của atof; tất cả kiểm tra tham số bị tắt. Ý nghĩa đặc biệt này của danh sách tham số trống được thiết kế để cho phép các chương trình C cũ được biên dịch với các trình biên dịch mới. Nhưng việc sử dụng nó với các chương trình mới là một ý tưởng tồi. Nếu hàm có tham số, hãy khai báo chúng; nếu nó không có tham số, hãy sử dụng **void.**

Vì vậy, đây là cách mà nguyên mẫu hàm của bạn sẽ trông như thế này:

|  |
| --- |
| int foo(void); |

Và đây là cách định nghĩa hàm sẽ là:

|  |
| --- |
| int foo(void) {  ...  <các câu lệnh>  ...  return 1;  } |

Một lợi ích của việc sử dụng cách trên, so với cách định nghĩa hàm foo() (tức là không sử dụng từ khóa "void"), là trình biên dịch có thể phát hiện lỗi nếu bạn gọi hàm của bạn bằng câu lệnh lỗi như foo(42). Loại câu lệnh gọi hàm này sẽ không gây ra lỗi nào nếu bạn để danh sách tham số trống. Lỗi sẽ diễn ra một cách yên lặng, không phát hiện và mã vẫn sẽ thực thi.

Điều này cũng có nghĩa là bạn nên định nghĩa hàm main() như sau:

|  |
| --- |
| int main(void)  {  ...  <các câu lệnh>  ...  return 0;  } |

Lưu ý rằng ngay cả khi hàm được định nghĩa với danh sách tham số trống, nó không cung cấp nguyên mẫu cho hàm, vì vậy trình biên dịch sẽ không cảnh báo nếu hàm sau đó được gọi với các đối số. Ví dụ:

|  |
| --- |
| #include <stdio.h>  static void parameterless(void)  {  printf("%s called\n", \_\_func\_\_);  }  int main(void)  {  parameterless(3, "arguments", "provided");  return 0;  } |

Nếu mã đó được lưu trong tệp proto79.c, nó có thể được biên dịch trên Unix với GCC (phiên bản 7.1.0 trên macOS Sierra 10.12.5 được sử dụng cho mục đích thử nghiệm) như sau:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -pedantic proto79.c -o proto79  $ |

Nếu bạn biên dịch với các tùy chọn nghiêm ngặt hơn, bạn sẽ nhận được lỗi:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-styledefinition -pedantic proto79.c -o proto79  proto79.c:3:13: error: function declaration isn’t a prototype [-Werror=strict-prototypes]  static void parameterless()  ^~~~~~~~~~  proto79.c: In function ‘parameterless’:  proto79.c:3:13: error: old-style function definition [-Werror=old-style-definition]  cc1: all warnings being treated as errors  $ |

Nếu bạn đặt cho hàm nguyên mẫu một tham số prototype static void parameterless(void), thì quá trình biên dịch sẽ cho lỗi:

|  |
| --- |
| $ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes -Wold-styledefinition -pedantic proto79.c -o proto79  proto79.c: In function ‘main’:  proto79.c:10:5: error: too many arguments to function ‘parameterless’  parameterless(3, "arguments", "provided");  ^~~~~~~~~  proto79.c:3:13: note: declared here  static void parameterless(void)  ^~~~~~~~~~~~~  $ |

Moral: Luôn luôn đảm bảo bạn có nguyên mẫu, và đảm bảo trình biên dịch của bạn cho bạn biết khi bạn không tuân thủ các quy tắc.

# NHỮNG LỖI PHỔ BIẾN

Phần này thảo luận về một số lỗi phổ biến mà một lập trình viên C nên biết và nên tránh mắc phải. Để biết thêm về một số sự cố không mong muốn và nguyên nhân của chúng, vui lòng xem **Undefined behavior**

## KẾT HỢP SINGED VÀ UNSIGNED INTEGERS TRONG CÁC PHÉP TOÁN SỐ HỌC

Thường thì không phải là một ý tưởng tốt khi kết hợp các số nguyên có dấu và không dấu trong các phép tính. Ví dụ, kết quả sẽ là gì trong ví dụ sau?

|  |
| --- |
| #include <stdio.h>  int main(void)  {  unsigned int a = 1000;  signed int b = -1;  if (a > b) puts("a is more than b");  else puts("a is less or equal than b");  return 0;  } |

Với giá trị 1000 lớn hơn -1, bạn sẽ hy vọng kết quả là a is more than b, tuy nhiên điều đó sẽ không đúng.

Các phép tính toán giữa các kiểu nguyên khác nhau được thực hiện trong một kiểu chung được xác định bởi những sự chuyển đổi thông thường của phép toán (xem cấu hình ngôn ngữ, 6.3.1.8).

Trong trường hợp này, "kiểu chung" là unsigned int, bởi vì, như được nêu trong ***Usual arithmetic conversions,***

714 Nếu phép toán có kiểu số nguyên không dấu có thứ hạng lớn hơn hoặc bằng thứ hạng của kiểu của phép toán khác, thì phép toán có kiểu số nguyên có dấu được chuyển đổi sang kiểu của phép toán có kiểu số nguyên không dấu.

Điều này có nghĩa rằng phần tử int b sẽ được chuyển đổi thành unsigned int trước khi thực hiện so sánh.

Khi -1 được chuyển đổi thành unsigned int, kết quả là giá trị unsigned int lớn nhất có thể, lớn hơn 1000, có nghĩa là a > b là sai.

## MACROS CHỈ LÀ SỰ THAY THẾ CHUỖI ĐƠN GIẢN

Macro là sự thay thế đơn giản cho chuỗi. (Nó hoạt động với các thông số tiền xử lý, không phải là chuỗi tùy ý.)

|  |
| --- |
| #include <stdio.h>  #define SQUARE(*x*) x\*x  int main(void) {  printf("%d\n", SQUARE(1+2));  return 0;  } |

Bạn có thể mong đợi đoạn mã này in ra 9 (3\*3), nhưng thực tế sẽ in ra 5 vì macro sẽ được mở rộng thành 1+2\*1+2.

Bạn nên bọc các đối số và toàn bộ biểu thức của macro trong dấu ngoặc đơn để tránh vấn đề này.

|  |
| --- |
| #include <stdio.h>  #define SQUARE(*x*) ((x)\*(x))  int main(void) {  printf("%d\n", SQUARE(1+2));  return 0;  } |

Một vấn đề khác là các đối số của macro không được đảm bảo được đánh giá một lần; chúng có thể không được đánh giá hoặc có thể được đánh giá nhiều lần.

|  |
| --- |
| #include <stdio.h>  #define MIN(*x*, *y*) ((x) <= (y) ? (x) : (y))  int main(void) {  int a = 0;  printf("%d\n", MIN(a++, 10));  printf("a = %d\n", a);  return 0;  } |

Trong đoạn mã này, macro sẽ được mở rộng thành ((a++) <= (10) ? (a++) : (10)). Vì a++ (0) nhỏ hơn 10, a++ sẽ được đánh giá hai lần và giá trị của a và giá trị được trả về từ MIN sẽ khác với bạn mong đợi.

Điều này có thể được tránh bằng cách sử dụng các hàm, nhưng lưu ý rằng các kiểu sẽ được cố định bởi định nghĩa hàm, trong khi macro có thể (quá) linh hoạt về kiểu.

|  |
| --- |
| #include <stdio.h>  int min(int *x*, int *y*) {  return *x* <= *y* ? *x* : *y*;  }  int main(void) {  int a = 0;  printf("%d\n", min(a++, 10));  printf("a = %d\n", a);  return 0;  } |

Bây giờ vấn đề của đánh giá hai lần đã được khắc phục, nhưng hàm min này không thể xử lý dữ liệu kiểu double mà không bị cắt xén, ví dụ.

Các chỉ thị macro có thể có hai loại:

|  |
| --- |
| #define OBJECT\_LIKE\_MACRO tiếp theo là một "danh sách thay thế" của mã thông báo tiền xử lý.  #define FUNCTION\_LIKE\_MACRO(with, arguments) tiếp theo là một danh sách thay thế |

Điều phân biệt hai loại macro này là ký tự theo sau sau bộ nhận dạng sau #define: nếu đó là dấu ngoặc tròn, đó là một macro dạng hàm; nếu không, đó là một macro dạng đối tượng. Nếu ý định là viết một macro dạng hàm, không được có khoảng trắng nào giữa cuối tên của macro và (. Kiểm tra điều này để có một giải thích chi tiết.

Phiên bản ≥ C99

Trong C99 hoặc mới hơn, bạn có thể sử dụng static inline int min(int x, int y) { … }.

Phiên bản ≥ C11

Trong C11, bạn có thể viết một biểu thức 'type-generic' cho min.

|  |
| --- |
| #include <stdio.h>  #define min(*x*, *y*) \_Generic((x), \  long double: min\_ld, \  unsigned long long: min\_ull, \  default: min\_i \  )(x, y)  #define gen\_min(*suffix*, *type*) \  static inline type min\_##suffix(type x, type y) { return (x < y) ? x : y; }  gen\_min(ld, long *double*)  gen\_min(ull, unsigned long *long*)  gen\_min(i, int)  int main(void)  {  unsigned long long ull1 = 50ULL;  unsigned long long ull2 = 37ULL;  printf("min(%llu, %llu) = %llu\n", ull1, ull2, min(ull1, ull2));  long double ld1 = 3.141592653L;  long double ld2 = 3.141592652L;  printf("min(%.10Lf, %.10Lf) = %.10Lf\n", ld1, ld2, min(ld1, ld2));  int i1 = 3141653;  int i2 = 3141652;  printf("min(%d, %d) = %d\n", i1, i2, min(i1, i2));  return 0;  } |

The generic expression có thể được mở rộng với nhiều kiểu khác nhau như double, float, long long, unsigned long, long, unsigned - và các lời gọi macro gen\_min thích hợp được viết.

## QUÊN SAO CHÉP GIÁ TRỊ TRẢ VỀ CỦA REALLOC VÀO MỘT BIẾN TẠM

Khi realloc thất bại, nó trả về NULL. Nếu bạn gán giá trị của bộ đệm ban đầu cho giá trị trả về của realloc và nếu nó trả về NULL, thì bộ đệm ban đầu (con trỏ cũ) bị mất, dẫn đến **memory leak.**Giải pháp là sao chép vào một con trỏ tạm, và nếu con trỏ tạm không phải là NULL, sau đó sao chép vào bộ đệm thực sự.

|  |
| --- |
| char \*buf, \*tmp;  buf = malloc(...);  ...  /\* SAI \*/  if ((buf = realloc(buf, 16)) == NULL)  perror("realloc");  /\* ĐÚNG \*/  if ((tmp = realloc(buf, 16)) != NULL)  buf = tmp;  else  perror("realloc"); |

Trong trường hợp realloc thất bại, giá trị của biến tạm tmp vẫn giữ nguyên bộ nhớ cũ, giúp bạn không mất dữ liệu và tránh **memory leak.**

## QUÊN CẤP PHÁT THÊM MỘT BYTE CHO \0

Khi bạn sao chép một chuỗi vào một bộ đệm đã cấp phát bằng malloc, luôn nhớ thêm 1 vào strlen.

|  |
| --- |
| char \*dest = malloc(strlen(src)); /\* SAI \*/  char \*dest = malloc(strlen(src) + 1); /\* ĐÚNG \*/  strcpy(*dest*, *src*); |

Điều này là vì strlen không bao gồm ký tự \0 cuối cùng trong chiều dài. Nếu bạn tiếp cận SAI (như được hiển thị ở trên), khi gọi strcpy, chương trình của bạn sẽ gây ra hành vi không xác định.

Điều này cũng áp dụng trong tình huống bạn đang đọc một chuỗi có độ dài tối đa đã biết từ stdin hoặc một nguồn khác. Ví dụ:

|  |
| --- |
| #define MAX\_INPUT\_LEN 42  char buffer[MAX\_INPUT\_LEN]; /\* SAI \*/  char buffer[MAX\_INPUT\_LEN + 1]; /\* ĐÚNG \*/  scanf("%42s", buffer); /\* Đảm bảo rằng bộ đệm không bị tràn \*/ |

## HIỂU SAI VỀ SỰ ARRAY DECAY

Một vấn đề phổ biến trong mã sử dụng các mảng đa chiều, mảng các con trỏ, v.v. là sự thật rằng Type\*\* và Type[M][N] là các kiểu khác nhau:

|  |
| --- |
| #include <stdio.h>  void print\_strings(char \*\**strings*, *size\_t* *n*)  {  *size\_t* i;  for (i = 0; i < *n*; i++)  puts(*strings*[i]);  }  int main(void)  {  char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};  print\_strings(s, 4);  return 0;  } |

Kết quả mẫu từ trình biên dịch:

|  |
| --- |
| file1.c: In function 'main':  file1.c:13:23: error: passing argument 1 of 'print\_strings' from incompatible pointer type [- Wincompatible-pointer-types]  print\_strings(strings, 4);  ^  file1.c:3:10: note: expected 'char \*\*' but argument is of type 'char (\*)[20]'  void print\_strings(char \*\*strings, size\_t n) |

Lỗi cho biết rằng mảng s trong hàm main được chuyển đến hàm print\_strings, mà mong đợi một kiểu con trỏ khác với kiểu nó nhận được. Nó cũng bao gồm lưu ý về kiểu mà print\_strings mong đợi và kiểu đã truyền vào từ main.

Vấn đề xuất phát từ việc gọi là phân rã mảng. Khi s với kiểu char[4][20] (mảng của 4 mảng có 20 ký tự) được truyền vào hàm, nó sẽ biến thành một con trỏ đến phần tử đầu tiên của nó như thể bạn đã viết &s[0], kiểu char (\*)[20] (con trỏ đến 1 mảng có 20 ký tự). Điều này xảy ra cho bất kỳ mảng nào, bao gồm cả mảng con trỏ, mảng các mảng của mảng (mảng 3 chiều) và mảng các con trỏ đến mảng. Dưới đây là một bảng minh họa về việc gì xảy ra khi một mảng phân rã. Nhấn mạnh các thay đổi trong mô tả kiểu để minh họa việc gì xảy ra:

|  |  |  |  |
| --- | --- | --- | --- |
| trước khi phân rã |  | sau khi phân rã |  |
| char [20] | mảng (20 ký tự) | char \* | con trỏ tới (1 ký tự) |
| char [4][20] | mảng của (4 mảng 20 ký tự) | char (\*)[20] | con trỏ tới (1 mảng 20 ký tự) |
| char \*[4] | mảng (4 con trỏ tới 1 ký tự) | char \*\* | con trỏ tới (1 con trỏ tới 1 ký tự) |
| char [3][4][20] | mảng của (3 mảng 4 mảng 20 ký tự) | char (\*)[4][20] | con trỏ tới (1 mảng 4 mảng 20 ký tự) |
| char (\*[4])[20] | mảng (4 con trỏ tới 1 mảng 20 ký tự) | char (\*\*)[20] | con trỏ tới (1 con trỏ tới 1 mảng 20 ký tự) |

Nếu một mảng có thể phân rã thành con trỏ, thì có thể nói rằng con trỏ có thể được coi là một mảng có ít nhất 1 phần tử. Một ngoại lệ cho điều này là con trỏ null, trỏ đến không gì và do đó không phải là một mảng.

Sự phân rã mảng chỉ xảy ra một lần. Nếu một mảng đã phân rã thành con trỏ, thì nó đã trở thành một con trỏ, không phải là mảng. Ngay cả khi bạn có một con trỏ đến một mảng, hãy nhớ rằng con trỏ có thể được xem xét là một mảng có ít nhất một phần tử, vì vậy sự phân rã mảng đã xảy ra.

Nói cách khác, một con trỏ đến một mảng (char (\*)[20]) sẽ không bao giờ trở thành một con trỏ đến một con trỏ (char \*\*). Để sửa hàm print\_strings, chỉ cần làm cho nó nhận kiểu đúng:

|  |
| --- |
| void print\_strings(char (\**strings*)[20], *size\_t* *n*)  /\* HOẶC \*/  void print\_strings(char strings[][20], size\_t *n*) |

Một vấn đề nảy sinh khi bạn muốn hàm print\_strings trở nên tổng quát cho bất kỳ mảng ký tự nào: nếu có 30 ký tự thay vì 20? Hoặc 50? Câu trả lời là thêm một tham số khác trước tham số mảng:

|  |
| --- |
| #include <stdio.h>  /\*  \* Lưu ý về việc sắp xếp lại các tham số và thay đổi tên tham số  \* từ các định nghĩa trước đó:  \* n (số lượng chuỗi)  \* => scount (số lượng chuỗi)  \*  \* Tất nhiên, bạn cũng có thể sử dụng một trong các dạng được khuyến nghị dưới đây  \* cho tham số `strings` thay thế:  \*  \* char strings[scount][ccount]  \* char strings[][ccount]  \*/  void in\_cac\_chuoi(size\_t *so\_luong\_chuoi*, size\_t *do\_dai\_chuoi*, char (\*chuoi)[do\_dai\_chuoi])  {  *size\_t* i;  for (i = 0; i < so\_luong\_chuoi; i++)  puts(chuoi[i]);  }  int main(void)  {  char s[4][20] = {"Example 1", "Example 2", "Example 3", "Example 4"};  in\_cac\_chuoi(4, 20, s);  return 0;  } |

Biên dịch nó sẽ không gây ra lỗi và cho ra kết quả như mong đợi:

|  |
| --- |
| Example 1  Example 2  Example 3  Example 4 |

## QUÊN GIẢI PHÓNG BỘ NHỚ (MEMORY LEAKS)

Một thực hành tốt trong lập trình là giải phóng bất kỳ bộ nhớ nào đã được cấp phát trực tiếp bởi mã của bạn, hoặc ngầm định thông qua việc gọi các hàm nội bộ hoặc ngoại vi, như một API thư viện như strdup(). Quên giải phóng bộ nhớ có thể dẫn đến việc xảy ra rò rỉ bộ nhớ, dẫn đến việc tích luỹ một lượng lớn bộ nhớ bị lãng phí mà không sử dụng, có thể gây ra sự cố hoặc hành vi không xác định. Các vấn đề có khả năng xảy ra nếu việc rò rỉ được xảy ra lặp đi lặp lại trong một vòng lặp hoặc hàm đệ quy. Nguy cơ của việc gặp sự cố trong chương trình tăng lên khi chương trình rò rỉ chạy càng lâu. Đôi khi các vấn đề có thể xuất hiện ngay lập tức; lúc khác, vấn đề có thể không thấy trong vài giờ hoặc ngay cả sau nhiều năm hoạt động liên tục. Sự cố do cạn kiệt bộ nhớ có thể gây ra hậu quả nghiêm trọng, tùy thuộc vào tình huống.

Vòng lặp vô hạn dưới đây là một ví dụ về việc rò rỉ bộ nhớ\*\* có thể dần tiêu thụ bộ nhớ có sẵn bằng cách gọi hàm getline(), một hàm tạo bộ nhớ mới ngầm định mà không giải phóng bộ nhớ đó.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  int main(void)  {  char \*line = NULL;  *size\_t* size = 0;    /\* Vòng lặp dưới đây rò rỉ bộ nhớ càng nhanh càng tốt \*/  for (;;)  {  getline(&line, &size, stdin); /\* Bộ nhớ mới được cấp phát ngầm định \*/  /\* <làm bất kỳ điều gì> \*/  line = NULL;  }  return 0;  } |

Ngược lại, đoạn mã dưới đây cũng sử dụng hàm getline(), nhưng lần này, bộ nhớ đã được cấp phát được giải phóng đúng cách, tránh rò rỉ.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  int main(void)  {  char \*line = NULL;  *size\_t* size = 0;  for (;;)  {  if (getline(&line, &size, stdin) < 0)  {  free(line);  line = NULL;  /\* Xử lý lỗi như đặt cờ, thoát khỏi vòng lặp và/hoặc kết thúc chương trình \*/  }  /\* <làm bất kỳ điều gì> \*/  free(line);  line = NULL;  }  return 0;  } |

Rò rỉ bộ nhớ không luôn luôn có hậu quả rõ ràng và không nhất thiết là một vấn đề chức năng. Mặc dù "thực hành tốt" đề ra là giải phóng bộ nhớ một cách nghiêm ngặt tại các điểm và điều kiện chiến lược, nhằm giảm bớt dấu chân bộ nhớ và giảm nguy cơ cạn kiệt bộ nhớ, nhưng cũng có những ngoại lệ. Ví dụ, nếu chương trình có thời gian hoạt động và phạm vi giới hạn, nguy cơ gặp sự cố cấp phát có thể được coi là quá nhỏ để lo lắng. Trong trường hợp đó, việc bỏ qua việc giải phóng rõ ràng có thể được coi là chấp nhận được. Ví dụ, hầu hết các hệ điều hành hiện đại tự động giải phóng toàn bộ bộ nhớ mà chương trình tiêu thụ khi chương trình kết thúc, cho dù đó là do chương trình gặp lỗi, cuộc gọi hệ thống đến hàm exit(), chấm dứt tiến trình hoặc đạt đến cuối main(). Việc giải phóng bộ nhớ một cách rõ ràng tại thời điểm kết thúc chương trình có thể gây ra tình trạng lặp lại hoặc giới hạn hiệu suất.

Việc cấp phát có thể thất bại nếu bộ nhớ không đủ, và việc xử lý sự cố nên được xem xét ở mức độ phù hợp trên ngăn xếp cuộc gọi. Hàm getline(), được hiển thị ở trên, là một tình huống sử dụng thú vị vì nó là một hàm thư viện không chỉ cấp phát bộ nhớ mới mà còn để cho người gọi giải phóng, nhưng có thể thất bại vì nhiều lý do khác nhau, tất cả những lý do đó đều phải được xem xét. Do đó, khi sử dụng một API C, đọc tài liệu (trang man) và chú ý đặc biệt đến các điều kiện lỗi và việc sử dụng bộ nhớ, và biết rõ lớp phần mềm nào chịu trách nhiệm giải phóng bộ nhớ được trả về.

Một thực hành xử lý bộ nhớ khác thường là đặt các con trỏ bộ nhớ thành NULL ngay sau khi bộ nhớ mà con trỏ đó tham chiếu đã được giải phóng, để các con trỏ này có thể được kiểm tra tính hợp lệ bất kỳ lúc nào (ví dụ: kiểm tra NULL / không NULL), bởi vì truy cập vào bộ nhớ đã giải phóng có thể dẫn đến các vấn đề nghiêm trọng như lấy dữ liệu vô nghĩa (phép đọc), hoặc hỏng dữ liệu (phép ghi) và/hoặc làm cho chương trình bị sập. Trong hầu hết các hệ điều hành hiện đại, việc giải phóng vị trí bộ nhớ 0 (NULL) không có tác dụng (ví dụ: nó là vô hại), như được yêu cầu bởi tiêu chuẩn C — vì vậy bằng cách thiết lập một con trỏ thành NULL, không có rủi ro của việc giải phóng bộ nhớ hai lần nếu con trỏ được chuyển cho hàm free(). Hãy nhớ rằng việc giải phóng bộ nhớ hai lần có thể dẫn đến những vấn đề mất thời gian, khó hiểu và khó chẩn đoán.

## SAO CHÉP QUÁ NHIỀU

|  |
| --- |
| char buf[8]; /\* tiny buffer, easy to overflow \*/  printf("What is your name?\n");  scanf("%s", buf); /\* WRONG \*/  scanf("%7s", buf); /\* RIGHT \*/ |

Nếu người dùng nhập vào một chuỗi dài hơn 7 ký tự (-1 đối với ký tự kết thúc null), bộ nhớ phía sau bộ đệm buf sẽ bị ghi đè. Điều này dẫn đến hành vi không xác định. Tin tặc độc hại thường khai thác điều này để ghi đè địa chỉ trả lại và thay đổi nó thành địa chỉ chứa mã độc của tin tặc.

## NHẦM LẪN VIẾT = THAY VÌ == KHI SO SÁNH

Toán tử = được sử dụng để gán giá trị.

Toán tử == được sử dụng để so sánh.

Mọi người cần cẩn trọng để không nhầm lẫn hai toán tử này. Đôi khi người ta viết sai như sau:

|  |
| --- |
| /\* gán y cho x \*/  if (x = y) {  /\* logic \*/  } |

trong khi thực ra điều mong muốn là:

|  |
| --- |
| /\* so sánh x có bằng y không \*/  if (x == y) {  /\* logic \*/  } |

Phần đầu gán giá trị của y cho x và kiểm tra xem giá trị đó có khác 0 hay không, thay vì thực hiện so sánh. Điều này tương đương với:

|  |
| --- |
| if ((x = y) != 0) {  /\* logic \*/  } |

Có những trường hợp khi kiểm tra kết quả của việc gán giá trị được dự định và thường được sử dụng, vì nó tránh việc phải nhân đôi mã và phải xử lý lần đầu một cách đặc biệt. So sánh:

|  |
| --- |
| while ((c = getopt\_long(argc, argv, short\_options, long\_options, &*option\_index*)) != -1) {  switch (c) {  ...  }  } |

Với

|  |
| --- |
| while ((c = getopt\_long(argc, argv, short\_options, long\_options, &*option\_index*)) != -1) {  switch (c) {  ...  }  } |

Các trình biên dịch hiện đại sẽ nhận ra mẫu này và không cảnh báo khi việc gán được đặt trong dấu ngoặc như trên, nhưng có thể cảnh báo cho các sử dụng khác. Ví dụ:

|  |
| --- |
| if (x = y) /\* cảnh báo \*/  if ((x = y)) /\* không cảnh báo \*/  if ((x = y) != 0) /\* không cảnh báo; tường minh \*/ |

Một số lập trình viên sử dụng chiến lược đặt hằng số ở bên trái của toán tử (thường được gọi là điều kiện Yoda). Bởi vì hằng số là lvalue, kiểu điều kiện này sẽ khiến trình biên dịch báo lỗi nếu sử dụng sai toán tử.

|  |
| --- |
| if (5 = y) /\* Lỗi \*/  if (5 == y) /\* Không lỗi \*/ |

Tuy nhiên, cách này làm giảm tính đọc của mã và không được coi là cần thiết nếu lập trình viên tuân theo các nguyên tắc lập trình C tốt, và không giúp khi so sánh hai biến nên nó không phải là một giải pháp phổ quát. Hơn nữa, nhiều trình biên dịch hiện đại có thể cảnh báo khi mã được viết với các điều kiện Yoda.

## KÍ TỰ XUỐNG DÒNG KHÔNG ĐƯỢC SƯ DỤNG TRONG LỜI GỌI SCANF() THÔNG THƯỜNG

Khi chương trình sau đây được thực thi với đầu vào như sau:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {  int num = 0;  char str[128], \*lf;  scanf("%d", &num);  fgets(str, sizeof(str), stdin);    if ((lf = strchr(str, '\n')) != NULL) \*lf = '\0';  printf("%d \"%s\"\n", num, str);  return 0;  } |

và đầu vào là:

|  |
| --- |
| 42  life |

đầu ra sẽ là 42 "" thay vì kết quả mong đợi là 42 "life".

Lý do là do ký tự xuống dòng sau số 42 không được sử dụng trong lời gọi scanf() và nó lại được sử dụng bởi fgets() trước khi đọc chuỗi "life". Sau đó, fgets() ngừng đọc trước khi đọc "life".

Để tránh vấn đề này, một cách hữu ích khi độ dài tối đa của dòng là đã biết - ví dụ khi giải quyết các vấn đề trong hệ thống trọng tài trực tuyến - là tránh sử dụng scanf() trực tiếp và đọc tất cả các dòng qua fgets(). Bạn có thể sử dụng sscanf() để phân tích các dòng đã đọc.

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {  int num = 0;  char line\_buffer[128] = "", str[128], \*lf;  fgets(line\_buffer, sizeof(line\_buffer), stdin);  sscanf(line\_buffer, "%d", &num);  fgets(str, sizeof(str), stdin);    if ((lf = strchr(str, '\n')) != NULL) \*lf = '\0';  printf("%d \"%s\"\n", num, str);  return 0;  } |

Một cách khác là đọc cho đến khi bạn gặp một ký tự xuống dòng sau khi sử dụng scanf() và trước khi sử dụng fgets().

|  |
| --- |
| #include <stdio.h>  #include <string.h>  int main(void) {  int num = 0;  char str[128], \*lf;  int c;  scanf("%d", &num);  while ((c = getchar()) != '\n' && c != EOF);  fgets(str, sizeof(str), stdin);    if ((lf = strchr(str, '\n')) != NULL) \*lf = '\0';  printf("%d \"%s\"\n", num, str);  return 0;  } |

## THÊM DẤU CHẤM PHẨY VÀO #DEFINE

Dễ dàng bị nhầm lẫn trong trình xử lý tiền xử lý C và xem nó như một phần của C chính, nhưng đó là sai lầm vì trình xử lý tiền xử lý chỉ là cơ chế thay thế văn bản. Ví dụ, nếu bạn viết

|  |
| --- |
| #define MAX 100;  int arr[MAX]; |

mã sẽ mở rộng thành

|  |
| --- |
| int arr[100;]; |

điều này là một lỗi cú pháp. Giải pháp là loại bỏ dấu chấm phẩy khỏi dòng #define. Gần như luôn luôn là một lỗi khi kết thúc #define bằng dấu chấm phẩy.

## SỬ DỤNG DẤU CHẤM PHẨY KHÔNG CẨN THẬN

Hãy cẩn thận khi sử dụng dấu chấm phẩy. Ví dụ sau đây

|  |
| --- |
| if (x > a);  a = x; |

thực chất có nghĩa là:

|  |
| --- |
| if (x > a) {}  a = x; |

điều này có nghĩa là x sẽ được gán cho a trong mọi trường hợp, điều đó có thể không phải là điều bạn mong muốn ban đầu.

Đôi khi, thiếu dấu chấm phẩy cũng có thể gây ra vấn đề khó phát hiện:

|  |
| --- |
| if (i < 0)  return  day = date[0];  hour = date[1];  minute = date[2]; |

Dấu chấm phẩy sau return đã bị bỏ sót, vì vậy day=date[0] sẽ được trả về.

Một kỹ thuật để tránh vấn đề này và các vấn đề tương tự là luôn sử dụng dấu ngoặc nhọn cho các điều kiện và vòng lặp có nhiều dòng. Ví dụ:

|  |
| --- |
| if (x > a) {  a = x;  } |

## LỖI UNDEFINED REFERENCE KHI LINKING

Một trong những lỗi phổ biến nhất trong quá trình biên dịch xảy ra trong giai đoạn linking. Lỗi có dạng tương tự như sau:

|  |
| --- |
| $ gcc undefined\_reference.c  /tmp/ccoXhwF0.o: In function `main':  undefined\_reference.c:(.text+0x15): undefined reference to `foo'  collect2: error: ld returned 1 exit status  $ |

Vậy, hãy xem xét mã đã tạo ra lỗi này:

|  |
| --- |
| int foo(void);  int main(int *argc*, char \*\**argv*)  {  int foo\_val;  foo\_val = foo();  return foo\_val;  } |

Ở đây, chúng ta thấy một sự khai báo của foo (int foo();), nhưng không có định nghĩa thực tế cho nó (hàm thực sự). Vì vậy, chúng ta cung cấp trình biên dịch với phần khai báo hàm, nhưng không có hàm như vậy được định nghĩa ở bất kỳ đâu. Do đó, giai đoạn biên dịch diễn ra mà không có vấn đề gì, nhưng trình liên kết dừng với lỗi "Undefined reference".

Để sửa lỗi này trong chương trình nhỏ của chúng ta, chúng ta chỉ cần thêm một định nghĩa cho foo:

|  |
| --- |
| /\* Khai báo foo \*/  int foo(void);  /\* Định nghĩa foo \*/  int foo(void)  {  return 5;  }  int main(int *argc*, char \*\**argv*)  {  int foo\_val;  foo\_val = foo();  return foo\_val;  } |

Bây giờ mã này sẽ được biên dịch thành công. Trường hợp thay thế khác xảy ra khi mã nguồn cho foo() nằm trong một tệp nguồn riêng foo.c (và có một tiêu đề foo.h để khai báo foo() và được bao gồm cả trong foo.c và undefined\_reference.c). Lúc này, để sửa lỗi, chúng ta cần liên kết cả tệp đối tượng từ foo.c và undefined\_reference.c, hoặc biên dịch cả hai tệp nguồn:

|  |
| --- |
| $ gcc -c undefined\_reference.c  $ gcc -c foo.c  $ gcc -o working\_program undefined\_reference.o foo.o |

Hoặc

|  |
| --- |
| $ gcc -o working\_program undefined\_reference.c foo.c |

Một tình huống phức tạp hơn là khi liên quan đến thư viện, như trong mã sau:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <math.h>  int main(int *argc*, char \*\**argv*)  {  double first;  double second;  double power;  if (*argc* != 3)  {  fprintf(stderr, "Usage: %s <denom> <nom>\n", *argv*[0]);  return EXIT\_FAILURE;  }  /\* Chuyển đổi đầu vào người dùng thành số, kiểm tra lỗi bổ sung  \* nên được thực hiện ở đây. \*/  first = strtod(*argv*[1], NULL);  second = strtod(*argv*[2], NULL);  /\* Sử dụng hàm pow() từ libm - điều này sẽ gây ra lỗi liên kết  \* trừ khi mã này được biên dịch với libm! \*/  power = pow(first, second);    printf("%f mũ %f = %f\n", first, second, power);  return EXIT\_SUCCESS;  } |

Mã này cú pháp đúng, khai báo cho pow() tồn tại từ #include <math.h>, vì vậy chúng ta cố gắng biên dịch và liên kết (link) nhưng nhận được lỗi như sau:

|  |
| --- |
| $ gcc no\_library\_in\_link.c -o no\_library\_in\_link  /tmp/ccduQQqA.o: In **function** `main':  **no\_library\_in\_link.c:(.text+0x8b): undefined reference to `pow'**  collect2: error: ld returned 1 exit status  $ |

Điều này xảy ra vì trình liên kết không tìm thấy định nghĩa cho pow() trong giai đoạn liên kết. Để sửa lỗi này, chúng ta cần chỉ định rằng chúng ta muốn liên kết với thư viện toán học libm bằng cách chỉ định cờ -lm. (Lưu ý rằng có các nền tảng như macOS mà không cần -lm, nhưng khi bạn gặp lỗi tham chiếu chưa xác định, thì thư viện này cần thiết.)

Vì vậy, chúng ta thực hiện giai đoạn biên dịch lại, lần này chỉ định thư viện (sau các tệp nguồn hoặc đối tượng):

|  |
| --- |
| $ gcc no\_library\_in\_link.c -lm -o library\_in\_link\_cmd  $ ./library\_in\_link\_cmd 2 4  2.000000 mũ 4.000000 = 16.000000 |

Và nó hoạt động

## KIỂM TRA BIỂU THỨC LOGIC VỚI TRUE

Tiêu chuẩn C ban đầu không có kiểu dữ liệu Boolean nội tại, vì vậy bool, **true** và **false** không có ý nghĩa cố định và thường được định nghĩa bởi các lập trình viên. Thông thường, **true** sẽ được định nghĩa là 1 và **false** sẽ được định nghĩa là 0.

Phiên bản ≥ C99

C99 thêm kiểu dữ liệu tích hợp \_Bool và tiêu đề **<stdbool.h>** định nghĩa bool (mở rộng thành \_Bool), **false** và **true**. Nó cũng cho phép bạn định nghĩa lại bool, **true** và **false**, nhưng ghi chú rằng đây là tính năng sắp cải tổ.

Quan trọng hơn, các biểu thức logic xử lý bất cứ thứ gì có giá trị 0 như là false và bất kỳ giá trị khác không bằng 0 như là true. Ví dụ:

|  |
| --- |
| /\* Trả về 'true' nếu bit quan trọng nhất được đặt \*/  bool isUpperBitSet(uint8\_t *bitField*)  {  if ((*bitField* & 0x80) == true) /\* Phép so sánh chỉ thành công nếu true là 0x80 và bitField có  bit đó được đặt \*/  {  return true;  }  else  {  return false;  }  } |

Trong ví dụ trên, hàm đang kiểm tra xem bit quan trọng nhất có được đặt và trả về true nếu có. Tuy nhiên, bằng cách kiểm tra một cách rõ ràng với true, câu lệnh if chỉ sẽ thành công nếu (bitfield & 0x80) được đánh giá thành true, có nghĩa là true được định nghĩa như là 1 và rất ít khi là 0x80. Hãy kiểm tra một cách rõ ràng cho trường hợp bạn mong đợi:

|  |
| --- |
| /\* Trả về 'true' nếu bit quan trọng nhất được đặt \*/  bool isUpperBitSet(uint8\_t *bitField*)  {  if ((*bitField* & 0x80) == 0x80) /\* Kiểm tra rõ ràng cho trường hợp chúng ta mong đợi \*/  {  return true;  }  else  {  return false;  }  } |

Hoặc đánh giá bất kỳ giá trị không bằng 0 nào như là true.

|  |
| --- |
| /\* Trả về 'true' nếu bit quan trọng nhất được đặt \*/  bool isUpperBitSet(uint8\_t *bitField*)  {  /\* Nếu bit quan trọng nhất được đặt, kết quả là 0x80 mà if sẽ đánh giá là true \*/  if (*bitField* & 0x80)  {  return true;  }  else  {  return false;  }  } |

## THỰC HIỆN MỞ RỘNG THÊM TRONG CON TRỎ SỐ HỌC

Trong phép tính con trỏ, số nguyên được thêm hoặc trừ vào con trỏ được hiểu không phải là sự thay đổi địa chỉ mà là số phần tử cần di chuyển.

|  |
| --- |
| #include <stdio.h>  int main(void) {  int array[] = {1, 2, 3, 4, 5};  int \*ptr = &array[0];  int \*ptr2 = ptr + sizeof(int) \* 2; /\* sai \*/  printf("%d %d\n", \*ptr, \*ptr2);  return 0;  } |

Mã này thực hiện mở rộng thêm trong việc tính toán con trỏ được gán cho ptr2. Nếu sizeof(int)là 4, điển hình trong môi trường 32 bit hiện đại, biểu thức này là viết tắt của "8 phần tử sau array[0]", nằm ngoài phạm vi và nó gọi hành vi không xác định .

Để ptr2 trỏ tới vị trí 2 phần tử sau array[0], bạn chỉ cần thêm 2.

|  |
| --- |
| #include <stdio.h>  int main(void) {  int array[] = {1, 2, 3, 4, 5};  int \*ptr = &array[0];  int \*ptr2 = ptr + 2;  printf("%d %d\n", \*ptr, \*ptr2); /\* "1 3" sẽ được in ra \*/  return 0;  } |

Phép tính con trỏ rõ ràng bằng các toán tử cộng có thể gây nhầm lẫn, vì vậy sử dụng chỉ số mảng có thể tốt hơn.

|  |
| --- |
| #include <stdio.h>  int main(void) {  int array[] = {1, 2, 3, 4, 5};  int \*ptr = &array[0];  int \*ptr2 = &ptr[2];  printf("%d %d\n", \*ptr, \*ptr2); /\* "1 3" sẽ được in ra \*/  return 0;  } |

E1[E2] tương đồng với (\*((E1)+(E2))) (N1570 6.5.2.1, đoạn 2), và &(E1[E2]) tương đương với ((E1)+(E2)) (N1570 6.5.3.2, ghi chú 102).

Hoặc nếu thích sử dụng toán tử con trỏ, việc chuyển đổi con trỏ để trỏ tới một kiểu dữ liệu khác có thể cho phép địa chỉ byte. Tuy nhiên, cần thận trọng: : **endianness** có thể trở thành vấn đề, và chuyển đổi sang các kiểu dữ liệu khác ngoài 'con trỏ tới ký tự' có thể dẫn đến **strict aliasing problems**

|  |
| --- |
| #include <stdio.h>  int main(void) {  int array[3] = {1,2,3}; // 4 bytes \* 3 allocated  unsigned char \*ptr = (unsigned char \*) array; // unsigned chars only take 1 byte  /\*  \* Now any pointer arithmetic on ptr will match  \* bytes in memory. ptr can be treated like it  \* was declared as: unsigned char ptr[12];  \*/  return 0;  } |

## CHÚ THÍCH NHIỀU DÒNG KHÔNG THỂ LỒNG NHAU

Trong ngôn ngữ C, bình luận nhiều dòng bằng /\* và \*/ không thể lồng nhau.

Nếu bạn chú thích một khối mã hoặc hàm bằng kiểu bình luận này:

|  |
| --- |
| /\*  \* max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  \* Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  \* arr: Mảng các số nguyên cần tìm.  \* num: Số lượng số nguyên trong arr.  \*/  int max(int *arr*[], int *num*)  {  int max = *arr*[0];  for (int i = 0; i < *num*; i++)  if (*arr*[i] > max)  max = *arr*[i];  return max;  } |

Bạn sẽ không thể dễ dàng chú thích nó ra:

|  |
| --- |
| // Cố gắng chú thích khối...  /\*  /\*  \* max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  \* Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  \* arr: Mảng các số nguyên cần tìm.  \* num: Số lượng số nguyên trong arr.  \*/  int max(int *arr*[], int *num*)  {  int max = *arr*[0];  for (int i = 0; i < *num*; i++)  if (*arr*[i] > max)  max = *arr*[i];  return max;  }  // Gây lỗi ở dòng dưới...  \*/ |

Một giải pháp là sử dụng kiểu chú thích của C99:

|  |
| --- |
| // max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  // Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  // arr: Mảng các số nguyên cần tìm.  // num: Số lượng số nguyên trong arr.  int max(int *arr*[], int *num*)  {  int max = *arr*[0];  for (int i = 0; i < *num*; i++)  if (*arr*[i] > max)  max = *arr*[i];  return max;  } |

Bây giờ, toàn bộ khối mã có thể được chú thích ra dễ dàng:

|  |
| --- |
| /\*  // max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  // Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  // arr: Mảng các số nguyên cần tìm.  // num: Số lượng số nguyên trong arr.  int max(int arr[], int num)  {  int max = arr[0];  for (int i = 0; i < num; i++)  if (arr[i] > max)  max = arr[i];  return max;  }  \*/ |

Một giải pháp khác là tránh vô hiệu hóa mã bằng cách sử dụng các chỉ thị tiền xử lý #ifdef hoặc #ifndef thay vì cú pháp bình luận. Các chỉ thị này có khả năng lồng nhau, để bạn tự do chú thích mã của bạn theo kiểu bạn ưa thích.

|  |
| --- |
| #define DISABLE\_MAX /\* Xóa hoặc chú thích dòng này để bật khối mã max() \*/  #ifdef DISABLE\_MAX  /\*  \* max(): Tìm số nguyên lớn nhất trong một mảng và trả về nó.  \* Nếu độ dài mảng nhỏ hơn 1, kết quả là không xác định.  \* arr: Mảng các số nguyên cần tìm.  \* num: Số lượng số nguyên trong arr.  \*/  int max(int *arr*[], int *num*)  {  int max = *arr*[0];  for (int i = 0; i < *num*; i++)  if (*arr*[i] > max)  max = *arr*[i];  return max;  }  #endif |

Một số hướng dẫn thậm chí khuyến nghị rằng các phần mã không bao giờ được chú thích và nếu mã phải tạm thời bị vô hiệu hóa, bạn có thể sử dụng chỉ thị tiền xử lý #if 0.

Xem #if 0 để tắt các phần code

## BỎ QUA GIÁ TRỊ TRẢ VỀ CỦA CÁC HÀM THƯ VIỆN

Hầu hết mọi hàm trong thư viện chuẩn C đều trả về một giá trị thành công và một giá trị lỗi. Ví dụ, hàm malloc sẽ trả về con trỏ đến khối bộ nhớ được cấp phát bởi hàm khi thành công và, nếu hàm không thể cấp phát khối bộ nhớ yêu cầu, trả về con trỏ null. Do đó, bạn luôn nên kiểm tra giá trị trả về để dễ dàng gỡ lỗi hơn.

Đây là một ví dụ không tốt:

|  |
| --- |
| char\* x = malloc(100000000000UL \* sizeof \**x*);  /\* thêm code \*/  scanf("%s", x); /\* Điều này có thể gây ra hành vi không xác định và nếu may mắn gây lỗi  vi phạm phân đoạn, trừ khi hệ thống của bạn có nhiều bộ nhớ \*/ |

Đây là một ví dụ tốt:

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  int main(void)  {  char\* x = malloc(100000000000UL \* sizeof \*x);  if (x == NULL) {  perror("malloc() failed");  exit(EXIT\_FAILURE);  }  if (scanf("%s", x) != 1) {  fprintf(stderr, "could not read string\n");  free(x);  exit(EXIT\_FAILURE);  }  /\* Thực hiện công việc với x. \*/  /\* Dọn dẹp. \*/  free(x);  return EXIT\_SUCCESS;  } |

Bằng cách này, bạn biết ngay lập tức nguyên nhân gây ra lỗi, nếu không bạn có thể dành hàng giờ để tìm kiếm lỗi ở một nơi hoàn toàn sai.

## SO SÁNH SỐ DẤU CHẤM ĐỘNG

Các loại số dấu chấm động (float, double và long double) không thể biểu diễn chính xác một số số do chúng có độ chính xác hữu hạn và biểu diễn các giá trị trong một định dạng nhị phân. Tương tự như chúng ta có các số thập phân lặp lại trong cơ số 10 cho các phân số như 1/3, cũng có các phân số không thể biểu diễn được theo cách hữu hạn trong cơ số nhị phân (như 1/3, nhưng quan trọng hơn là 1/10). Không nên so sánh trực tiếp các giá trị số dấu chấm động; hãy sử dụng một giá trị delta thay thế.

|  |
| --- |
| #include <float.h> // cho DBL\_EPSILON và FLT\_EPSILON  #include <math.h> // cho fabs()  int main(void)  {  double a = 0.1; // không chính xác: (nhị phân) 0.000110...  // có thể sai hoặc đúng  if (a + a + a + a + a + a + a + a + a + a == 1.0) {  printf("10 \* 0.1 thực sự là 1.0. Điều này không được đảm bảo trong trường hợp tổng quát.\n");  }  // Sử dụng một giá trị delta nhỏ.  if (fabs(a + a + a + a + a + a + a + a + a + a - 1.0) < 0.000001) {  // C99 5.2.4.2.2p8 đảm bảo ít nhất 10 chữ số thập phân  // của độ chính xác cho kiểu double.  printf("10 \* 0.1 gần như là 1.0.\n");  }  return 0;  } |

Ví dụ khác

|  |
| --- |
| #include <stdio.h>  #include <math.h>  static inline double rel\_diff(double *a*, double *b*)  {  return fabs(*a* - *b*) / fmax(fabs(*a*), fabs(*b*));  }  int main(void)  {  double d1 = 3.14159265358979;  double d2 = 355.0 / 113.0;    double epsilon = 1.0;  for (int i = 0; i < 10; i++)  {  if (rel\_diff(d1, d2) < epsilon)  printf("%d:%.10f <=> %.10f trong ngưỡng %.10f (sự chênh lệch tương đối %.4E)\n",  i, d1, d2, epsilon, rel\_diff(d1, d2));  else  printf("%d:%.10f <=> %.10f vượt quá ngưỡng %.10f (sự chênh lệch tương đối %.4E)\n",  i, d1, d2, epsilon, rel\_diff(d1, d2));  epsilon /= 10.0;  }  return 0;  } |

Kết quả

|  |
| --- |
| 0:3.1415926536 <=> 3.1415929204 trong ngưỡng 1.0000000000 (sự chênh lệch tương đối 8.4914E-08)  1:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.1000000000 (sự chênh lệch tương đối 8.4914E-08)  2:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0100000000 (sự chênh lệch tương đối 8.4914E-08)  3:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0010000000 (sự chênh lệch tương đối 8.4914E-08)  4:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0001000000 (sự chênh lệch tương đối 8.4914E-08)  5:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0000100000 (sự chênh lệch tương đối 8.4914E-08)  6:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0000010000 (sự chênh lệch tương đối 8.4914E-08)  7:3.1415926536 <=> 3.1415929204 trong ngưỡng 0.0000001000 (sự chênh lệch tương đối 8.4914E-08)  8:3.1415926536 <=> 3.1415929204 vượt quá ngưỡng 0.0000000100 (sự chênh lệch tương đối 8.4914E-08)  9:3.1415926536 <=> 3.1415929204 vượt quá ngưỡng 0.0000000010 (sự chênh lệch tương đối 8.4914E-08) |

## CHỮ SỐ THẬP PHÂN DẤU CHẤM ĐỘNG MẶC ĐỊNH LÀ KIỂU DOUBLE

Khi làm việc với các chữ số thập phân dấu chấm động, cần lưu ý rằng mặc định chúng là kiểu double. Điều này có thể dẫn đến hành vi không mong muốn khi khởi tạo biến kiểu float bằng các giá trị chữ số thập phân dấu chấm động hoặc so sánh chúng với các giá trị chữ số thập phân dấu chấm động.

Ví dụ:

|  |
| --- |
| #include <stdio.h>  int main() {  float n;  n = 0.1;  if (n > 0.1) printf("Wierd\n");  return 0;  }  // Prints "Wierd" when n is float |

Ở đây, nđược khởi tạo và làm tròn thành độ chính xác duy nhất, dẫn đến giá trị 0,10000000149011612. Sau đó, nđược chuyển đổi trở lại độ chính xác kép để so sánh với 0.1chữ (tương đương với 0,10000000000000001), dẫn đến sự không khớp.

Bên cạnh lỗi làm tròn, việc trộn floatcác biến với doublechữ sẽ dẫn đến hiệu suất kém trên các nền tảng không có hỗ trợ phần cứng cho độ chính xác kép.

## SỬ DỤNG HẰNG KÍ TỰ THAY VÌ CHUỖI KÍ TỰ VÀ NGƯỢC LẠI

Trong ngôn ngữ C, hằng ký tự và chuỗi ký tự là hai khái niệm khác nhau.

Hằng ký tự, được bao quanh bởi dấu nháy đơn như 'a', là một hằng số ký tự. Một hằng ký tự là một số nguyên có giá trị là mã ký tự đại diện cho ký tự tương ứng. Cách giải thích hằng ký tự chứa nhiều ký tự như 'abc' là được xác định bởi triển khai của ngôn ngữ.

Một chuỗi ký tự là một dãy ký tự được bao quanh bởi dấu nháy kép như "abc". Chuỗi ký tự là một mảng không thể thay đổi với các phần tử kiểu char. Chuỗi trong dấu nháy kép kèm theo ký tự null kết thúc là nội dung của chuỗi, do đó "abc" có 4 phần tử ({'a', 'b', 'c', '\0'}).

Trong ví dụ này, hằng ký tự được sử dụng trong trường hợp nên sử dụng chuỗi ký tự. Hằng ký tự này sẽ được chuyển đổi thành một con trỏ theo cách được xác định bởi triển khai của ngôn ngữ và khó có cơ hội để con trỏ đã chuyển đổi này hợp lệ, do đó ví dụ này sẽ gây ra lỗi hành vi không xác định.

|  |
| --- |
| #include <stdio.h>  int main(void) {  const char \*hello = 'hello, world'; /\* sai \*/  puts(hello);  return 0;  } |

Trong ví dụ này, một chuỗi ký tự được sử dụng trong trường hợp nên sử dụng hằng ký tự. Con trỏ được chuyển đổi từ chuỗi ký tự này sẽ được chuyển đổi thành một số nguyên theo cách được xác định bởi triển khai của ngôn ngữ, và sau đó sẽ được chuyển đổi thành char theo cách được xác định bởi triển khai của ngôn ngữ. (Cách chuyển đổi một số nguyên thành kiểu có dấu không thể biểu diễn giá trị cần chuyển đổi là do triển khai của ngôn ngữ quy định, và liệu kiểu char có phải là kiểu có dấu không cũng là do triển khai quy định.) Kết quả xuất ra sẽ là một cái gì đó vô nghĩa.

|  |
| --- |
| #include <stdio.h>  int main(void) {  char c = "a"; /\* sai \*/  printf("%c\n", c);  return 0;  } |

Hầu hết các trình biên dịch đều sẽ phát ra cảnh báo về các lỗi này. Nếu trình biên dịch không cảnh báo, bạn cần sử dụng nhiều tùy chọn cảnh báo hơn, hoặc đề xuất sử dụng trình biên dịch tốt hơn.

## HÀM ĐỆ QUY – THIẾU ĐIỀU KIỆN CƠ SỞ

Việc tính giai thừa của một số là một ví dụ cổ điển về hàm đệ quy.

**Thiếu Điều Kiện Cơ Sở:**

|  |
| --- |
| #include <stdio.h>  int factorial(int *n*)  {  return *n* \* factorial(*n* - 1);  }  int main()  {  printf("Giai thừa của %d = %d\n", 3, factorial(3));  return 0;  } |

Đầu ra điển hình:**Segmentation fault: 11**

Vấn đề với chức năng này là nó sẽ lặp vô hạn, gây ra lỗi phân đoạn — nó cần một điều kiện cơ sở để dừng đệ quy.

**Điều kiện cơ sở được khai báo:**

|  |
| --- |
| #include <stdio.h>  int factorial(int *n*)  {  if (*n* == 1) // Điều kiện cơ sở, rất quan trọng trong thiết kế hàm đệ quy.  {  return 1;  }  else  {  return *n* \* factorial(*n* - 1);  }  }  int main()  {  printf("Giai thừa của %d = %d\n", 3, factorial(3));  return 0;  } |

Kết quả mẫu:

|  |
| --- |
| Giai thừa của 3 = 6 |

Hàm này sẽ kết thúc ngay khi đạt đến điều kiện n bằng 1 (miễn là giá trị ban đầu của n đủ nhỏ - giới hạn trên là 12 khi int có kích thước 32 bit).

**Các quy tắc cần tuân theo:**

1. Khởi tạo thuật toán. Chương trình đệ quy thường cần một giá trị khởi đầu để bắt đầu. Điều này có thể được thực hiện thông qua việc sử dụng tham số được truyền vào hàm hoặc thông qua việc cung cấp một hàm cổng không đệ quy nhưng thiết lập giá trị khởi đầu cho tính toán đệ quy.
2. Kiểm tra xem giá trị hiện tại đang xử lý có khớp với trường hợp cơ sở hay không. Nếu có, xử lý và trả về giá trị.
3. Định lại câu trả lời dưới dạng một vấn đề con nhỏ hơn hoặc đơn giản hơn.
4. Chạy thuật toán trên vấn đề con.
5. Kết hợp các kết quả để tạo thành câu trả lời.
6. Trả về kết quả.

Nguồn: Hàm Đệ Quy

## VƯỢT QUA GIỚI HẠN MẢNG

Mảng trong C được đánh chỉ mục bằng số 0, có nghĩa là chỉ mục luôn bắt đầu từ 0 và kết thúc với chỉ mục bằng độ dài của mảng trừ 1. Do đó, mã sau sẽ không xuất ra phần tử đầu tiên của mảng và sẽ xuất ra giá trị ngẫu nhiên cho giá trị cuối cùng mà nó in ra.

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int x = 0;  int myArray[5] = {1, 2, 3, 4, 5}; // Khai báo 5 phần tử  for(x = 1; x <= 5; x++) // Lặp từ 1 đến 5.  printf("%d\t", myArray[x]);  printf("\n");  return 0;  } |

Kết quả: 2 3 4 5 Giá\_trị\_ngẫu\_nhiên

Mã dưới đây mô tả cách đạt được kết quả mong muốn một cách chính xác:

|  |
| --- |
| #include <stdio.h>  int main(void)  {  int x = 0;  int myArray[5] = {1, 2, 3, 4, 5}; // Khai báo 5 phần tử  for(x = 0; x < 5; x++) // Lặp từ 0 đến 4.  printf("%d\t", myArray[x]);    printf("\n");  return 0;  } |

Kết quả: 1 2 3 4 5

Quan trọng là biết độ dài của mảng trước khi làm việc với nó, nếu không bạn có thể gây hỏng buffer hoặc gây ra lỗi chia cho không bằng cách truy cập vào các vị trí bộ nhớ nằm ngoài phạm vi.

## TRUYỀN MẢNG KHÔNG KỀ NHAU VÀO HÀM MONG ĐỢI MẢNG ĐA CHIỀU “THỰC SỰ”

Khi cấp phát các mảng đa chiều bằng malloc, calloc và realloc, một mẫu phổ biến là cấp phát các mảng con bằng nhiều lệnh gọi (ngay cả khi lệnh gọi chỉ xuất hiện một lần, nó có thể nằm trong vòng lặp):

|  |
| --- |
| /\* Cũng có thể là `int \*\*` với malloc được sử dụng để cấp phát mảng ngoại. \*/  int \*array[4];  int i;  /\* Cấp phát 4 mảng có 16 phần tử int. \*/  for (i = 0; i < 4; i++)  array[i] = malloc(16 \* sizeof(\*array[i])); |

Sự khác biệt về byte giữa phần tử cuối cùng của một trong các mảng con và phần tử đầu tiên của mảng con tiếp theo có thể không phải là 0 như chúng sẽ là với một mảng đa chiều "thực sự" (ví dụ: int array[4][16];):

|  |
| --- |
| /\* 0x40003c, 0x402000 \*/  printf("%p, %p\n", (void \*)(array[0] + 15), (void \*)array[1]); |

Tính cả kích thước của int, bạn có được sự khác biệt là 8128 byte (8132-4), tương đương với 2032 phần tử mảng có kích thước int, và đó là vấn đề: một mảng đa chiều "thực sự" không có khoảng trống giữa các phần tử.

Nếu bạn cần sử dụng một mảng được cấp phát động với một hàm mong đợi một mảng đa chiều "thực sự", bạn nên cấp phát một đối tượng có kiểu int \* và sử dụng toán học để thực hiện tính toán:

|  |
| --- |
| void func(int *M*, int *N*, int \**array*);  ...  int \*array;  int M = 4, N = 16;  array = calloc(M, N \* sizeof(\*array));  array[i \* N + j] = 1;  func(*M*, *N*, *array*); |

Nếu N là một macro hoặc một số nguyên nguyên thay vì một biến, mã có thể đơn giản sử dụng cú pháp mảng 2-D sau khi cấp phát con trỏ cho một mảng:

|  |
| --- |
| void func(int *M*, int *N*, int \**array*);  #define N 16  void func\_N(int *M*, int (\**array*)[N]);  ...  int M = 4;  int (\*array)[N];  array = calloc(M, sizeof(\*array));  array[i][j] = 1;  func(*M*, N, (int \*)*array*);  func\_N(*M*, *array*); |

Phiên bản ≥ C99

Nếu N không phải là một macro hoặc một số nguyên nguyên, thì array sẽ trỏ đến một mảng có độ dài biến (VLA). Điều này vẫn có thể sử dụng với func bằng cách ép kiểu thành int \* và một hàm mới func\_vla sẽ thay thế func\_N:

|  |
| --- |
| void func(int *M*, int *N*, int \**array*);  void func\_vla(int *M*, int *N*, int array[M][N]);  ...  int M = 4, N = 16;  int (\*array)[N];  array = calloc(M, sizeof(\*array));  array[i][j] = 1;  func(*M*, N, (int \*)*array*);  func\_vla(*M*, N, *array*); |

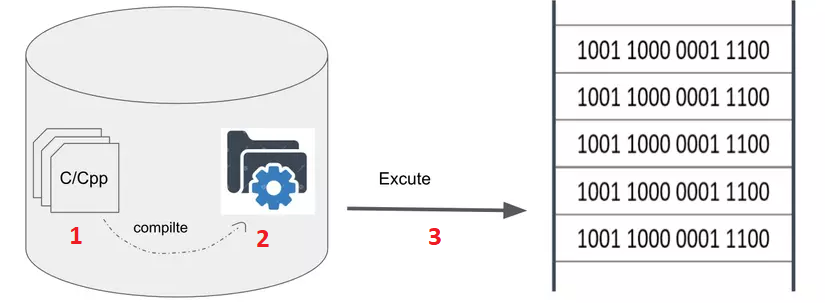
Phiên bản ≥ C11

Lưu ý: VLA là tùy chọn từ phiên bản C11. Nếu triển khai của bạn hỗ trợ C11 và x

# MEMORY LAYOUT

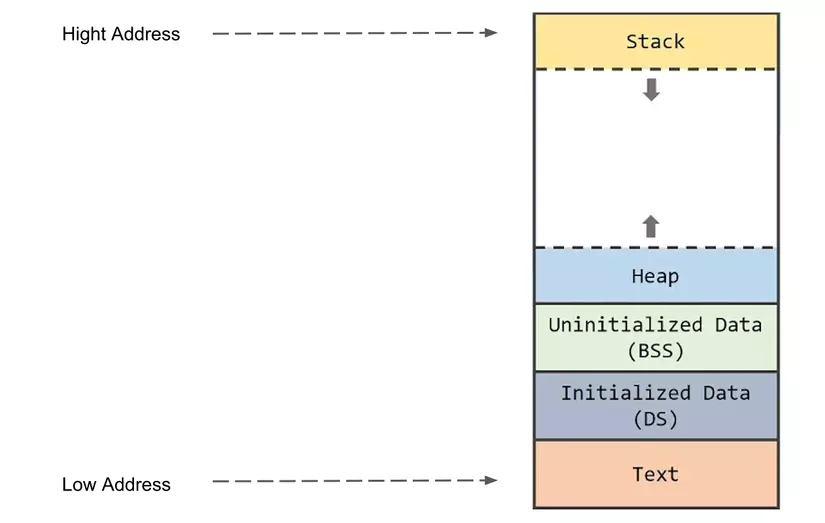
## Process In Program

1. Khi chúng ta viết 1 chương trình C/C++ , source code của chúng ta sẽ được lưu trong ổ cứng.
2. Sau đó khi chúng ta Compile source code (build), hệ thống sẽ tạo ra file cài đặt và file này cũng được lưu trong ổ cứng.
3. Cuối cùng là quá trình chúng ta run file program, thực chất đây là quá trình chúng ta load chương trình vào Memory và Excute.



**Hình 1**: Process Program

## Memory Layout In Program



**Hình 2**: Memory Layout In C

Memory layout của của một chương trình C/C++ gồm 5 phần chính: **Text Segment, Initialized Data Segment, Uninitialized Data Segment, Heap và Stack**

### Text Segment

Text Segment ở vùng nhớ của địa chỉ thấp nhất, mã máy trong trương trình sẽ được lưu trữ tại đây.

Vùng lưu trữ này chỉ có thể đọc.

### Initialized Data Segment (DS)

Initialized Data Segment ( Data Segment ) là nơi lưu trữ **global variables, static variables** với điều kiện các biến này được khởi tạo bởi programmer với **giá trị khác 0**.

**VD:**

|  |
| --- |
| int global = 100;  int foo() {  static int number = 10;  return 0;  } |
| Trong đoạn chương chính trên, biến global được khởi tạo với giá trị 100, và biến **static** number được khởi với giá trị 10 bởi programmer nên được lưu trữ vào Initialized Data Segment. |

### Uninitialized Data Segment (BSS)

Uninitialized Data Segment (BSS) là nơi lưu trữ **global variables, static variables** **không được khởi tạo** hoặc khởi tạo với **giá trị bằng 0.**

**VD:**

|  |
| --- |
| int global;  int foo() {  static int number = 0;  return 0;  } |
| Trong đoạn chương trình trên, biến global **không được khởi tạo giá trị** mặc định và biến **static** number **được khởi tạo với giá trị bằng 0** sẽ được lưu trữ vào Uninitialized Data Segment. |

### Heap (Dynamic Memory Allocation)

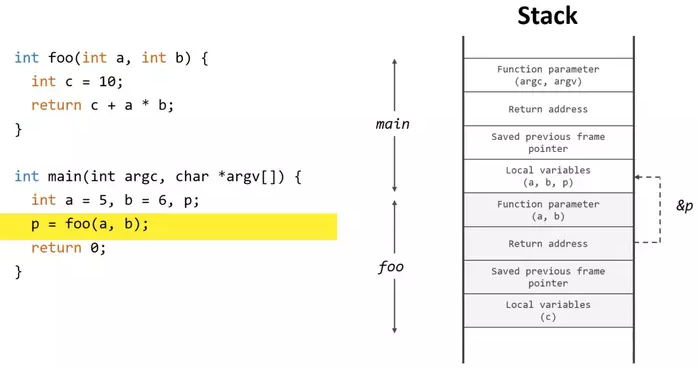
Trong C/C++ chúng ta có thể hoàn toàn control được quá trình cấp phát hoặc giải phóng bộ nhớ bằng các lệnh như malloc, calloc, relloc, free, new, delete, ... Vùng nhớ được cấp phát chính là HEAP, vùng nhớ này sẽ phình lên ( grows upward ) mỗi khi bạn cấp phát. Khi sử dụng xong các bạn phải delete vùng nhớ này. Nếu quên không delete sẽ gây ra hiện tượng Memory Leak.

### Stack (Automatic Variable Storage)

Stack là một vùng nhớ được cấp phát tự động và có cấu trúc LIFO (Last In First Out) để chứa các Function Frame. Mỗi khi chương trình được gọi, thì các Function Frame sẽ được gọi và push vào trong stack.

**Lưu ý**: Nếu chúng ta sử dụng hết vùng nhớ của Stack thì sẽ có một lỗi rất kinh điển đó là Stack OverFlow xảy ra.

**VD:** Function Frame có cấu trúc như ví dụ dưới đây.



Khi main() được gọi, function frame của main() sẽ được push vào stack, và khi function foo() được gọi thì function frame của foo() sẽ được push vào.

Cấu trúc của một function frame gồm bốn phần chính: Function Parameter, Return Address, Saved Previus Frame Pointer, Local Variable.

1. **Funtion Parameter** là các tham số truyền vào.
2. **Return Address** trong ví dụ trên, Return Address trả về địa chỉ p trong hàm main() chính là Local Variable.
3. **Saved Previus Frame Pointer** trỏ vào vị trí đầu tiên của function foo() tức là kết thúc của function main()
4. **Local Variable** là các biến local của function.