LẬP TRÌNH C

**MỤC LỤC**

[CHƯƠNG 1: MÔI TRƯỜNG 1](#_Toc164641063)

[1. GNU (GNU's Not Unix) 1](#_Toc164641064)

[2. GCC (GNU Compiler Collection) 1](#_Toc164641065)

[3. Trình biên dịch g++ 1](#_Toc164641066)

[4. Trình biên dịch gcc 2](#_Toc164641067)

[5. MinGW (Minimalist GNU for Windows) 2](#_Toc164641068)

[6. Cygwin 2](#_Toc164641069)

[CHƯƠNG 2: BUILD CHƯƠNG TRÌNH C/C++ BẰNG GCC 3](#_Toc164641070)

[1. Giới thiệu 3](#_Toc164641071)

[2. Kiểm tra hệ thống đã cài đặt trình biên dịch chưa? 3](#_Toc164641072)

[3. Biên dịch mã nguồn C/C++ với gcc hay g++ 4](#_Toc164641073)

[3.1. Lệnh build chương trình C bằng GCC 4](#_Toc164641074)

[3.2. Lệnh build chương trình C bằng G++ 4](#_Toc164641075)

[CHƯƠNG 3: BUILD CHƯƠNG TRÌNH C/C++ BẰNG GCC - NÂNG CAO 5](#_Toc164641076)

[1. Nhắc lại về kiến thức cũ 5](#_Toc164641077)

[2. Các bước để build 1 chương trình C/C++ 5](#_Toc164641078)

[3. Build module C/C++ với GCC 6](#_Toc164641079)

[4. Link thư viện với GCC 8](#_Toc164641080)

[4.1. Sơ lược thư viện liên kết tĩnh - Static Library 8](#_Toc164641081)

[4.2. Sơ lược thư viện liên kết động - Dynamic Library 8](#_Toc164641082)

[5. Bài tập 8](#_Toc164641083)

[CHƯƠNG 4: BẮT ĐẦU VỚI NGÔN NGỮ C 11](#_Toc164641084)

[1. Hello World 11](#_Toc164641085)

[2. Chỉnh sửa chương trình 13](#_Toc164641086)

[3. Biên dịch và chạy chương trình 13](#_Toc164641087)

[3.1. Biên dịch bằng GCC 13](#_Toc164641088)

[3.2. Sử dụng trình biên dịch clang 13](#_Toc164641089)

[3.3. Sử dụng trình biên dịch Microsoft C từ dòng lệnh 14](#_Toc164641090)

[3.4. Thực hiện chương trình 14](#_Toc164641091)

[4. “HELLO WORLD!” trong K&R C 14](#_Toc164641092)

[4.1. C90 §5.1.2.2.3 Kết thúc chương trình 15](#_Toc164641093)

[4.2. C90 §6.6.6.4 Câu lệnh return 15](#_Toc164641094)

[4.3. C99 §5.1.2.2.3 Kết thúc chương trình 15](#_Toc164641095)

[CHƯƠNG 5: CHÚ THÍCH 16](#_Toc164641096)

[1. Chú thích bằng cách sử dụng tiền xử lý (Preprocessor) 16](#_Toc164641097)

[2. Chú thích được giới hạn bằng /\* \*/ 16](#_Toc164641098)

[3. Chú thích được giới hạn bằng dấu // 18](#_Toc164641099)

[4. Rủi ro có thể xảy ra do TRIGRAPH 18](#_Toc164641100)

[CHƯƠNG 6: KIỂU DỮ LIỆU 20](#_Toc164641101)

[1. Giải thích các khai báo 20](#_Toc164641102)

[2. Kiểu số nguyên có độ rộng cố định ( kể từ c99) 23](#_Toc164641103)

[3. Kiểu số nguyên và hằng số 23](#_Toc164641104)

[4. Hằng số thực 26](#_Toc164641105)

[5. Chuỗi kí tự 26](#_Toc164641106)

[CHƯƠNG 7: TOÁN TỬ 29](#_Toc164641107)

[1. Toán tử quan hệ 29](#_Toc164641108)

[1.1. Bằng “==” 29](#_Toc164641109)

[1.2. Không bằng (Khác) “!=” 29](#_Toc164641110)

[1.3. Not “!” 30](#_Toc164641111)

[1.4. So sánh lớn hơn “>” 30](#_Toc164641112)

[1.5. So sánh bé hơn “<” 30](#_Toc164641113)

[1.6. So sánh lớn hơn hoặc bằng “>=” 30](#_Toc164641114)

[1.7. So sánh bé hơn hoặc bằng “<=” 31](#_Toc164641115)

[2. Toán tử điều kiện/ toán tử 3 ngôi 31](#_Toc164641116)

[3. Toán tử BITWISE 32](#_Toc164641117)

[4. Hoạt động ngắn mạch của toán tử logic 36](#_Toc164641118)

[5. Toán tử dấu phẩy 36](#_Toc164641119)

[6. Toán tử số học 36](#_Toc164641120)

[4.6.1 Số học cơ bản 36](#_Toc164641121)

[4.6.2 Toán tử cộng 37](#_Toc164641122)

[4.6.3 Toán tử trừ 37](#_Toc164641123)

[4.6.4 Toán tử nhân 37](#_Toc164641124)

[4.6.5 Toán tử chia 38](#_Toc164641125)

[4.6.6 Toán tử modulo (Phép chia lấy dư) 40](#_Toc164641126)

[4.6.7 Toán tử tăng giảm 40](#_Toc164641127)

[7. Toán tử truy cập 42](#_Toc164641128)

[4.7.1 Thuộc tính của đối tượng 42](#_Toc164641129)

[4.7.2 Thành viên của đối tượng được trỏ tới 42](#_Toc164641130)

[4.7.3 Lấy địa chỉ 43](#_Toc164641131)

[4.7.4 Dereference 44](#_Toc164641132)

[4.7.5 Chỉ số mảng 44](#_Toc164641133)

[4.7.6 Khả năng hoán đổi của chỉ số mảng 44](#_Toc164641134)

[8. Toán tử SIZEOF 45](#_Toc164641135)

[4.8.1 With a type as operand 45](#_Toc164641136)

[4.8.2 With an expression as operand 45](#_Toc164641137)

[9. Toán tử ép kiểu 45](#_Toc164641138)

[10. Toán tử gọi hàm 46](#_Toc164641139)

[11. TOÁN TỬ TĂNG/ TOÁN TỬ GIẢM 46](#_Toc164641140)

[12. Toán tử gán 47](#_Toc164641141)

[13. Toán tử logic 49](#_Toc164641142)

[4.13.1 Logic AND 49](#_Toc164641143)

[4.13.2 Logic OR 49](#_Toc164641144)

[4.13.3 Logic NOT 50](#_Toc164641145)

[14. TOÁN HỌC TRÊN CON TRỎ 51](#_Toc164641146)

[4.14.1 Cộng con trỏ 51](#_Toc164641147)

[4.14.2 Trừ con trỏ 51](#_Toc164641148)

[15. \_Alignof 51](#_Toc164641149)

[CHƯƠNG 8: BOOLEAN 52](#_Toc164641150)

[1. SỬ DỤNG STDBOOL.H 52](#_Toc164641151)

[2. Sử dụng #DEFINE 53](#_Toc164641152)

[3. SỬ DỤNG KIỂU \_BOOL NỘI TẠI(TÍCH HỢP) 54](#_Toc164641153)

[4. Số nguyên và con trỏ trong biểu thức BOOLEAN 55](#_Toc164641154)

[5. Xác định kiểu bool bằng cách sử dụng TYPEDEF 56](#_Toc164641155)

[CHƯƠNG 9: CHUỖI 58](#_Toc164641156)

[1. PHÂN TÁCH TỪ: STRTOK(), STRTOK\_R() VÀ STRTOK\_S() 58](#_Toc164641157)

[2. STRING LITERALS 61](#_Toc164641158)

[3. Tính độ dài chuỗi: STRLEN() 62](#_Toc164641159)

[4. Giới thiệu cơ bản về chuỗi 64](#_Toc164641160)

[5. SAO CHÉP CHUỖI 65](#_Toc164641161)

[5.1. Gán con trỏ không sao chép chuỗi 65](#_Toc164641162)

[5.2. Sao chép chuỗi dùng các hàm tiêu chuẩn 66](#_Toc164641163)

[5.2.1. strcpy() 66](#_Toc164641164)

[5.2.2. snprintf() 66](#_Toc164641165)

[5.2.3. strncat() 67](#_Toc164641166)

[5.2.4. strncpy() 68](#_Toc164641167)

[6. LẶP QUA CÁC KÍ TỰ TRONG MỘT CHUỖI 69](#_Toc164641168)

[7. Tạo mảng chuỗi 70](#_Toc164641169)

[8. Chuyển đổi chuỗi thành số: ATOI(), ATOF() (CẨN THẬN KHÔNG NÊN SỬ DỤNG) 71](#_Toc164641170)

[9. Đọc ghi dữ liệu dạng chuỗi 72](#_Toc164641171)

[10. TÌM VỊ TRÍ XUẤT HIỆN ĐẦU TIÊN/ CUỐI CÙNG CỦA MỘT KÍ TỰ CỤ THỂ: STRCHR(), STRRCHR() 73](#_Toc164641172)

[11. Sao chép và nối chuỗi: STRCPY(), STRCAT() 75](#_Toc164641173)

[12. So sánh: STRCMP(), STRNCMP(), STRCASECMP(), STRNCASECMP() 77](#_Toc164641174)

[13. Chuyển đổi chuỗi thành số một cách an toàn: Hàm STRTOX 80](#_Toc164641175)

[14. STRSPN và STRCSPN 82](#_Toc164641176)

[CHƯƠNG 10: CHỮ SỐ, KÝ TỰ VÀ CHUỖI 84](#_Toc164641177)

[1. FLOATING POINT LITERALS( CHỮ SỐ DẤU CHẤM ĐỘNG) 84](#_Toc164641178)

[2. STRING LITERALS( CHUỖI KÍ TỰ CHỮ) 84](#_Toc164641179)

[3. CHARACTER LITERALS(KÝ TỰ) 85](#_Toc164641180)

[4. INTEGER LITERALS(SỐ NGUYÊN) 86](#_Toc164641181)

[CHƯƠNG 11: COMPOUND LITERALS 88](#_Toc164641182)

[1. ĐỊNH NGHĨA/ KHỞI TẠO COMPOUND LITERALS 88](#_Toc164641183)

[CHƯƠNG 12: BIT- FIENDS 91](#_Toc164641184)

[CHƯƠNG 13: MẢNG 92](#_Toc164641185)

[1. Khai báo và khởi tạo mảng 92](#_Toc164641186)

[2. Lặp qua mảng hiệu quả và thứ tự ưu tiên hàng( row major order) 94](#_Toc164641187)

[3. Độ dài mảng 96](#_Toc164641188)

[4. Truyền mảng đa chiều vào hàm 98](#_Toc164641189)

[5. Mảng đa chiều 100](#_Toc164641190)

[5.1. Mảng hai chiều 100](#_Toc164641191)

[5.1.1. Khởi tạo mảng 2 chiều 101](#_Toc164641192)

[5.1.2. Truy cập phần tử trong mảng hai chiều 102](#_Toc164641193)

[5.2. Mảng ba chiều 103](#_Toc164641194)

[5.2.1. Bộ nhớ mảng ba chiều 103](#_Toc164641195)

[5.2.2. Khởi tạo một mảng ba chiều 103](#_Toc164641196)

[6. Define mảng và truy cập vào phần tử mảng 104](#_Toc164641197)

[7. Xóa nội dung của mảng( Thiết lập giá trị bằng 0) 105](#_Toc164641198)

[8. Thiết lập giá trị trong mảng 106](#_Toc164641199)

[9. Duyệt qua mảng bằng con trỏ 108](#_Toc164641200)

[CHƯƠNG 14: LINKED LISTS 110](#_Toc164641201)

[1. DOUBLY LINKED LIST (DANH SÁCH LIÊN KẾT KÉP) 110](#_Toc164641202)

[2. REVERSING A LINKED LIST (ĐẢO NGƯỢC DANH SÁCH LIÊN KẾT) 115](#_Toc164641203)

[3. INSERTING A NODE AT THE NTH POSITION (CHÈN NÚT Ở VỊ TRÍ THỨ N) 118](#_Toc164641204)

[4. INSERTING A NODE AT THE BEGINNING OF A SINGLY LINKED LIST (CHÈN NÚT VÀO ĐẦU DANH SÁCH LIÊN KẾT ĐƠN) 121](#_Toc164641205)

[CHƯƠNG 15: ENUMERATIONS 126](#_Toc164641206)

[1. SIMPLE ENUMERATION (ENUMERATION ĐƠN GIẢN) 126](#_Toc164641207)

[2. ENUMERATION CONSTANT WITHOUT TYPENAME (HẰNG SỐ ENUMERATION KHÔNG CẦN TÊN KIỂU) 128](#_Toc164641208)

[3. ENUMERATION WITH DUPLICATE VALUE (ENUMERATION VỚI GIÁ TRỊ TRÙNG LẶP) 128](#_Toc164641209)

[4. TYPEDEF ENUM 129](#_Toc164641210)

[CHƯƠNG 16: STRUCTS (CẤU TRÚC) 132](#_Toc164641211)

[1. FLEXIBLE ARRAY MEMBERS (THÀNH PHẦN MẢNG LINH HOẠT) 132](#_Toc164641212)

[1.1. Khai báo kiểu dữ liệu 132](#_Toc164641213)

[1.2. Ảnh hưởng đến kích thước và phần đệm 133](#_Toc164641214)

[1.3. Sử dụng 133](#_Toc164641215)

[1.4. Struct hack 135](#_Toc164641216)

[1.5. Khả năng tương thích 135](#_Toc164641217)

[2. TYPEDEF STRUCTS 137](#_Toc164641218)

[3. POINTERS TO STRUCTS (CON TRỎ TỚI CẤU TRÚC) 138](#_Toc164641219)

[4. PASSING STRUCTS TO FUNCTIONS (TRUYỀN STRUCTS VÀO HÀM) 142](#_Toc164641220)

[5. OBJECT- BASED PROGRAMMING USING STRUCTS (LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG SỬ DỤNG STRUCTS) 144](#_Toc164641221)

[6. SIMPLE DATA STRUCTURES (CÁC CẤU TRÚC DỮ LIỆU ĐƠN GIẢN) 148](#_Toc164641222)

[CHƯƠNG 17: STANDARD MATH (TOÁN HỌC TIÊU CHUẨN) 150](#_Toc164641223)

[1. CÁC HÀM MŨ – POW(), POWF(), POWL() 150](#_Toc164641224)

[2. DOUBLE PRECISION FLOATING- POINT REMAINDER FMOD() (PHẦN DƯ SỐ THỰC VỚI ĐỘ CHÍNH XÁC KÉP FMOD()) 152](#_Toc164641225)

[3. SINGLE PRECISION AND LONG DOUBLE PRECISION FLOATING- POINT REMAINDER: FMODF(), FMODL() 153](#_Toc164641226)

[CHƯƠNG 18: CÁC LỆNH LẶP/ VÒNG LẶP: FOR, WHILE, DO WHILE 155](#_Toc164641227)

[1. Vòng lặp FOR 155](#_Toc164641228)

[2. LOOP UNROLLING AND DUFF’S DEVICE 156](#_Toc164641229)

[3. VÒNG LẶP WHILE 157](#_Toc164641230)

[4. VÒNG LẶP DO – WHILE 158](#_Toc164641231)

[5. VÒNG LẶP VÔ HẠN 160](#_Toc164641232)

[CHƯƠNG 19: SELECTION STATEMENTS (CÂU LỆNH LỰA CHỌN) 162](#_Toc164641233)

[1. CÂU LỆNH IF() 162](#_Toc164641234)

[2. NESTED IF()… ELSE VS IF ELSE LADDER ( CÂU LỆNH IF LỒNG NHAU VÀ CÂU LỆNH IF BẬC THANG) 163](#_Toc164641235)

[3. CÂU LỆNH SWITCH() 165](#_Toc164641236)

[4. CÂU LỆNH IF()… ELSE VÀ CÚ PHÁP 169](#_Toc164641237)

[5. NỐI HAI HOẶC NHIỀU LỆNH IF()… ELSE 170](#_Toc164641238)

[CHƯƠNG 20: INITIALIZATION (KHỞI TẠO) 172](#_Toc164641239)

[1. KHỞI TẠO BIẾN TRONG C 172](#_Toc164641240)

[1.1. Khởi tạo một mảng 173](#_Toc164641241)

[1.2. 17.1.2 Khởi tạo mảng kí tự 173](#_Toc164641242)

[2. SỬ DỤNG DESIGNATED INITIALIZERS 174](#_Toc164641243)

[2.1. Designated initializers cho các phần tử mảng 174](#_Toc164641244)

[2.2. Designated initializers cho structures 175](#_Toc164641245)

[2.3. Designated initializer cho unions 175](#_Toc164641246)

[2.4. Designated initializers cho arrays của structures, etc 177](#_Toc164641247)

[2.5. Specifying ranges in array initializers (Chỉ định phạm vi trong bộ khởi tạo mảng) 178](#_Toc164641248)

[3. INITIALIZING STRUCTURES AND ARRAYS OF STRUCTURES (KHỞI TẠO CẤU TRÚC VÀ MẢNG CẤU TRÚC) 178](#_Toc164641249)

[CHƯƠNG 21: KHAI BÁO VÀ ĐỊNH NGHĨA 180](#_Toc164641250)

[1. Hiểu về khai báo và định nghĩa 180](#_Toc164641251)

[CHƯƠNG 22: COMMAND- LINE ARGUMENTS (ĐỐI SỐ DÒNG LỆNH) 182](#_Toc164641252)

[1. In ra các đối số của chương trình và chuyển đổi thành các giá trị số nguyên 182](#_Toc164641253)

[2. IN RA CÁC ĐỐI SỐ CỦA DÒNG LỆNH 183](#_Toc164641254)

[3. SỬ DỤNG CÔNG CỤ GETOPT CỦA GNU 184](#_Toc164641255)

[CHƯƠNG 23: FILES VÀ LUỒNG INPUT/OUTPUT 186](#_Toc164641256)

[1. Mở và ghi vào tệp tin 186](#_Toc164641257)

[1.1. Đóng một tập tin 189](#_Toc164641258)

[2. RUN PROCESS (QUÁ TRÌNH CHẠY) 189](#_Toc164641259)

[3. FPRINTF 190](#_Toc164641260)

[4. LẤY CÁC DÒNG TỪ TỆP SỬ DỤNG GETLINE() 191](#_Toc164641261)

[5. FSCANF() 197](#_Toc164641262)

[6. ĐỌC CÁC DÒNG TỪ TẬP TIN 199](#_Toc164641263)

[7. MỞ VÀ VIẾT VÀO FILE NHỊ PHÂN 201](#_Toc164641264)

[CHƯƠNG 24: FORMATTED INPUT/OUTPUT (ĐỊNH DẠNG ĐẦU VÀO ĐẦU RA) 205](#_Toc164641265)

[1. CÁC BỘ CHUYỂN ĐỔI CHO VIỆC IN RA 205](#_Toc164641266)

[2. HÀM PRINTF() 207](#_Toc164641267)

[3. CÁC CỜ ĐỊNH DẠNG KHI IN 208](#_Toc164641268)

[4. IN GIÁ TRỊ CỦA CON TRỎ TỚI ĐỐI TƯỢNG 210](#_Toc164641269)

[5. IN GIÁ TRỊ CỦA HIỆU HAI CON TRỎ ĐẾN MỘT ĐỐI TƯỢNG 212](#_Toc164641270)

[6. LENGTH MODIFIERS 213](#_Toc164641271)

[CHƯƠNG 25: CON TRỎ 216](#_Toc164641272)

[1. Giới thiệu 216](#_Toc164641273)

[2. Các lỗi thường gặp 220](#_Toc164641274)

[2.1. Không kiểm tra lỗi trong việc cấp phát bộ nhớ 220](#_Toc164641275)

[2.2. Sử dụng số nguyên cố định thay vì sizeof khi yêu cầu cấp phát bộ nhớ 220](#_Toc164641276)

[2.3. Rò rỉ bộ nhớ (Memory leaks) 221](#_Toc164641277)

[2.4. Lỗi Logic 221](#_Toc164641278)

[2.5. Tạo con trỏ trỏ tới biến được tạo trong vùng nhớ (stack) ngăn xếp 222](#_Toc164641279)

[2.6. Tăng giảm và dereferencing 225](#_Toc164641280)

[3. Dereferencing a pointer 225](#_Toc164641281)

[4. Dereferencing a pointer to a struct 226](#_Toc164641282)

[5. Con trỏ hằng 228](#_Toc164641283)

[5.1. Single Pointers 228](#_Toc164641284)

[5.2. Pointer to Pointer 229](#_Toc164641285)

[6. Con trỏ hàm 232](#_Toc164641286)

[7. Polymorphic behaviour with void pointers 234](#_Toc164641287)

[8. Toán tử địa chỉ 235](#_Toc164641288)

[9. Khởi tạo con trỏ 235](#_Toc164641289)

[10. Pointer to pointer 237](#_Toc164641290)

[11. Con trỏ void\* như là đối số và giá trị trả về của các hàm tiêu chuẩn 237](#_Toc164641291)

[12. Cùng một dấu hoa thị \* nhưng có ý nghĩa khác nhau 238](#_Toc164641292)

[CHƯƠNG 26: SEQUENCE POINTS 241](#_Toc164641293)

[1. BIỂU THỨC KHÔNG CÓ THỨ TỰ 241](#_Toc164641294)

[2. BIỂU THỨC CÓ THỨ TỰ 242](#_Toc164641295)

[3. BIỂU THỨC ĐƯỢC XÁC ĐỊNH THỨ TỰ MỘT CÁCH KHÔNG XÁC ĐỊNH 243](#_Toc164641296)

[CHƯƠNG 27: CON TRỎ HÀM 245](#_Toc164641297)

[1. GIỚI THIỆU 245](#_Toc164641298)

[2. TRẢ VỀ CON TRỎ HÀM TỪ MỘT HÀM 246](#_Toc164641299)

[3. THỰC HÀNH TỐT NHẤT 248](#_Toc164641300)

[4. GÁN CON TRỎ HÀM 250](#_Toc164641301)

[5. GHI NHỚ ĐỂ VIẾT CON TRỎ HÀM 251](#_Toc164641302)

[6. KHÁI NIỆM CƠ BẢN 252](#_Toc164641303)

[CHƯƠNG 28: THAM SỐ CỦA HÀM 255](#_Toc164641304)

[1. THAM SỐ ĐƯỢC TRUYỀN BẰNG GIÁ TRỊ (PASS BY VALUE) 255](#_Toc164641305)

[2. TRUYỀN MẢNG VÀO HÀM 256](#_Toc164641306)

[3. THỨ TỰ THỰC THI THAM SỐ TRONG HÀM 257](#_Toc164641307)

[4. SỬ DỤNG THAM SỐ CON TRỎ ĐỂ TRẢ VỀ NHIỀU GIÁ TRỊ 258](#_Toc164641308)

[5. VÍ DỤ VỀ HÀM TRẢ VỀ CẤU TRÚC CHỨA CÁC GIÁ TRỊ CÓ MÃ LỖI 258](#_Toc164641309)

[CHƯƠNG 29: TRUYỀN MỘT MẢNG 2D ĐẾN MỘT HÀM 261](#_Toc164641310)

[1. TRUYỀN MỘT MẢNG 2D ĐẾN MỘT HÀM 261](#_Toc164641311)

[2. SỬ DỤNG MẢNG PHẲNG LÀM MẢNG 2D 272](#_Toc164641312)

[CHƯƠNG 30: XỬ LÝ LỖI 275](#_Toc164641313)

[1. ERRNO 275](#_Toc164641314)

[2. STRERROR 275](#_Toc164641315)

[3. PERROR 276](#_Toc164641316)

[CHƯƠNG 31: UNDEFINED BEHAVIOR (HÀNH VI KHÔNG XÁC ĐỊNH) 277](#_Toc164641317)

[1. DEREFERENCING MỘT CON TRỎ TỚI BIẾN NGOÀI THỜI GIAN TỒN TẠI CỦA NÓ 277](#_Toc164641318)

[2. SAO CHÉP BỘ NHỚ CHỒNG CHÉO 278](#_Toc164641319)

[3. SIGNED INTEGER OVERFLOW (TRÀN SỐ NGUYÊN CÓ DẤU) 279](#_Toc164641320)

[4. SỬ DỤNG MỘT BIẾN CHƯA ĐƯỢC KHỞI TẠO 281](#_Toc164641321)

[5. DATA RACE 283](#_Toc164641322)

[6. ĐỌC GIÁ TRỊ CỦA CON TRỎ ĐÃ ĐƯỢC GIẢI PHÓNG 283](#_Toc164641323)

[7. SỬ DỤNG ĐỊNH DẠNG KHÔNG CHÍNH XÁC TRONG PRINTF 283](#_Toc164641324)

[8. SỬA ĐỔI CHUỖI KÝ TỰ 284](#_Toc164641325)

[9. TRUYỀN MỘT CON TRỎ NULL ĐỂ CHUYỂN ĐỔI PRINTF(%S) 284](#_Toc164641326)

[10. SỬA ĐỔI BẤT KÌ ĐỐI TƯỢNG NÀO NHIỀU LẦN GIỮA HAI SEQUENCE POINTS 285](#_Toc164641327)

[11. GIẢI PHÓNG BỘ NHỚ HAI LẦN 287](#_Toc164641328)

[12. DỊCH CHUYỂN BIT BẰNG CÁCH SỬ DỤNG SỐ ÂM HOẶC VƯỢT QUÁ CHIỀU RỘNG CỦA KIỂU DỮ LIỆU 287](#_Toc164641329)

[13. QUAY LẠI TỪ MỘT HÀM ĐƯỢC KHAI BÁO BẰNG BỘ XÁC ĐỊNH HÀM ‘\_NORETURN’ HOẶC ‘NORETURN’ 288](#_Toc164641330)

[14. TRUY CẬP BỘ NHỚ NGOÀI VÙNG NHỚ ĐƯỢC PHÂN BỔ 290](#_Toc164641331)

[15. THAY ĐỔI BIẾN CONST THÔNG QUA CON TRỎ 290](#_Toc164641332)

[16. ĐỌC MỘT ĐỐI TƯỢNG CHƯA KHỞI TẠO KHÔNG ĐƯỢC HỖ TRỢ BỞI BỘ NHỚ 291](#_Toc164641333)

[17. PHÉP CỘNG HOẶC PHÉP TRỪ CON TRỎ KHÔNG ĐƯỢC GIỚI HẠN ĐÚNG CÁCH 291](#_Toc164641334)

[18. DEREFERENCING CON TRỎ NULL 292](#_Toc164641335)

[19. SỬ DỤNG FFLUSH TRÊN LUỒNG ĐẦU VÀO 292](#_Toc164641336)

[20. LIÊN KẾT KHÔNG NHẤT QUÁN CỦA CÁC ĐỊNH DANH 293](#_Toc164641337)

[21. THIẾU CÂU LỆNH RETURN TRONG CÂU LỆNH TRẢ VỀ GIÁ TRỊ 294](#_Toc164641338)

[22. CHIA CHO SỐ 0 295](#_Toc164641339)

[23. CHUYỂN ĐỔI GIỮA CÁC LOẠI CON TRỎ TẠO RA KẾT QUẢ ĐƯỢC CĂN CHỈNH KHÔNG CHÍNH XÁC 296](#_Toc164641340)

[24. SỬA ĐỔI CHUỖI ĐƯỢC TRẢ VỀ BỞI CÁC HÀM GETENV, STRERROR, VÀ SETLOCALE 297](#_Toc164641341)

[CHƯƠNG 32: TẠO SỐ NGẪU NHIÊN 298](#_Toc164641342)

[1. TẠO SỐ NGẪU NHIÊN CƠ BẢN 298](#_Toc164641343)

[2. TRÌNH TẠO ĐỒNG DƯ ĐƯỢC HOÁN VỊ 299](#_Toc164641344)

[3. THẾ HỆ XORSHIFT 300](#_Toc164641345)

[4. HẠN CHẾ TẠO TRONG MỘT PHẠM VI NHẤT ĐỊNH 301](#_Toc164641346)

[CHƯƠNG 33: TIỀN XỬ LÝ VÀ MACRO 303](#_Toc164641347)

[1. HEADER INCLUDE GUARDS 303](#_Toc164641348)

[2. SỬ DỤNG #IF 0 ĐỂ CHẶN MÃ TRONG CÁC PHẦN 307](#_Toc164641349)

[3. MACROS CÓ HÌNH THỨC CỦA HÀM 308](#_Toc164641350)

[4. CHÈN CÁC TỆP NGUỒN 310](#_Toc164641351)

[5. CONDITIONAL INCLUSION AND CONDITIONAL FUNCTION SIGNATURE MODIFICATION 311](#_Toc164641352)

[6. \_ \_ CPLUSPLUS ĐỂ SỬ DỤNG CÁC PHẦN TỬ C NGOÀI MÃ C ++ KHI BIÊN DỊCH C++ BẰNG TÊN BỊ MÃ HÓA 315](#_Toc164641353)

[7. DÁN MÃ THÔNG BÁO 317](#_Toc164641354)

[8. MACRO ĐƯỢC XÁC ĐỊNH TRƯỚC 317](#_Toc164641355)

[8.1. Macro xác định trước bắt buộc 318](#_Toc164641356)

[8.2. Macro được xác định trước khác (không bắt buộc) 318](#_Toc164641357)

[9. MACRO VỚI ĐỐI SỐ BIẾN THIÊN 320](#_Toc164641358)

[10. THAY THẾ MACRO 321](#_Toc164641359)

[11. CHỈ THỊ LỖI 323](#_Toc164641360)

[12. KHAI TRIỂN FOREACH 323](#_Toc164641361)

# MÔI TRƯỜNG

GNU là dự án phần mềm miễn phí, cung cấp nhiều công cụ và phần mềm quan trọng như GCC (bao gồm 'gcc' và 'g++').

Trong đó MinGW và Cygwin là hai Gói môi trường phát triển phần mềm khác nhau trên Windows và Linux, đều sử dụng các công cụ từ dự án GNU, bao gồm như GCC, GDB, để biên dịch mã nguồn C/C++ thành chương trình thực thi tương ứng.

## GNU (GNU's Not Unix)

GNU (GNU's Not Unix) là một dự án phần mềm tự miễn phí, được khởi xướng bởi Richard Stallman vào năm 1983. Mục tiêu của dự án là phát triển một hệ điều hành giống Unix hoàn toàn miễn phí và mã nguồn mở. Dự án GNU cung cấp nhiều phần mềm quan trọng như:

* GCC (GNU Compiler Collection) - Bộ công cụ biên dịch cho nhiều ngôn ngữ lập trình
* GNU Binutils - Các công cụ để xử lý tệp đối tượng
* GNU Make - Công cụ tự động hóa việc biên dịch
* GNU Bash - Trình thông dịch dòng lệnh
* GNU Emacs - Trình soạn thảo văn bản
* GNU Debugger (GDB) - Trình gỡ lỗi
* GNU Core Utils - Các tiện ích cơ bản cho dòng lệnh
* Và nhiều phần mềm khác...

Tất cả các phần mềm của GNU đều có mã nguồn mở và được phát hành dưới Giấy phép Công cộng GNU (GNU General Public License - GPL).

## GCC (GNU Compiler Collection)

* GCC là một bộ công cụ biên dịch mã nguồn mở, được phát triển bởi dự án GNU.
* GCC hỗ trợ biên dịch cho nhiều ngôn ngữ lập trình khác nhau như C, C++, Fortran, Ada, Go, và nhiều ngôn ngữ khác.
* Trong GCC, có các trình biên dịch riêng cho từng ngôn ngữ lập trình, ví dụ như 'gcc' cho C và 'g++' cho C++.

### Trình biên dịch g++

* g++ là tên của trình biên dịch C++ trong bộ công cụ GNU Compiler Collection (GCC).
* g++ được sử dụng để biên dịch mã nguồn C++ thành mã máy có thể thực thi.
* Cả MinGW và Cygwin đều bao gồm g++ trong bộ công cụ của chúng.

### Trình biên dịch ****gcc****

* gcc là tên của trình biên dịch C trong bộ công cụ GNU Compiler Collection (GCC).
* gcc được sử dụng để biên dịch mã nguồn C thành mã máy có thể thực thi.
* Giống như g++, cả MinGW và Cygwin đều bao gồm gcc trong bộ công cụ của chúng.

## MinGW (Minimalist GNU for Windows)

* MinGW là một gói môi trường phát triển phần mềm miễn phí và mã nguồn mở cho Windows.
* Nó cung cấp một bộ công cụ nhỏ gọn bao gồm trình biên dịch GCC (đặc biệt là g++ và gcc), trình liên kết, thư viện và các công cụ hỗ trợ khác từ dự án GNU.
* Khi sử dụng MinGW, bạn có thể biên dịch mã nguồn C/C++ thành các chương trình thực thi năng (executable) cho Windows, không cần bất kỳ thành phần bổ sung nào khác.

## Cygwin

* Cygwin là một gói môi trường giả lập Unix/Linux cho Windows, cung cấp một lớp tương thích để chạy các ứng dụng Unix/Linux trên Windows.
* Cygwin bao gồm một bộ công cụ GNU đầy đủ, bao gồm trình biên dịch GCC, và cung cấp một môi trường dòng lệnh giống Unix.
* Ứng dụng được biên dịch bằng Cygwin yêu cầu cài đặt Cygwin trên máy chạy để có thể chạy đúng cách.
* Khi sử dụng Cygwin, bạn có thể biên dịch mã nguồn C/C++ thành các chương trình thực thi cho môi trường Cygwin, nhưng cần cài đặt Cygwin trên máy chạy để chương trình có thể hoạt động đúng cách.

## Thiết Lập Môi Trường

Để bắt đầu học lập trình C, bước đầu tiên là thiết lập một môi trường cho phép bạn nhập và chỉnh sửa chương trình bằng ngôn ngữ C, cùng với một trình biên dịch để xây dựng các tệp thực thi có thể chạy trên hệ điều hành của bạn. Bạn cần hai công cụ phần mềm trên máy tính của mình:

1. **Trình Biên Dịch C (C Compiler)**
2. **Trình Soạn Thảo Văn Bản (Text Editor)**

### Trình Biên Dịch C

Mã nguồn được viết trong tệp mã nguồn là mã mà con người có thể đọc được cho chương trình của bạn. Mã này cần được "biên dịch" (compiled) thành ngôn ngữ máy để CPU có thể thực thi chương trình theo các chỉ dẫn đã cho.

Có nhiều trình biên dịch C khác nhau, dưới đây là một số trình biên dịch C được sử dụng rộng rãi:

* **GNU Compiler Collection (GCC)**

GCC là một trình biên dịch C mã nguồn mở phổ biến. Nó có sẵn cho nhiều nền tảng bao gồm Windows, macOS và Linux. GCC nổi tiếng với nhiều tính năng đa dạng và hỗ trợ cho nhiều tiêu chuẩn của ngôn ngữ C.

* **Clang**  
  Clang là một trình biên dịch C mã nguồn mở, thuộc dự án LLVM. Nó có sẵn trên nhiều nền tảng bao gồm Windows, macOS và Linux. Clang nổi tiếng về tốc độ và khả năng tối ưu hóa.
* **Microsoft Visual C++**

Microsoft Visual C++ là một trình biên dịch C độc quyền được phát triển bởi Microsoft. Nó chỉ có sẵn trên hệ điều hành Windows. Visual C++ nổi tiếng với việc tích hợp với môi trường phát triển Microsoft Visual Studio.

* **Turbo C**

Turbo C là một trình biên dịch C đã ngừng phát triển, được phát triển bởi Borland. Nó từng phổ biến vào đầu những năm 1990, nhưng hiện nay không còn được sử dụng rộng rãi.

# NGÔN NGỮ LẬP TRÌNH C

## Ngôn ngữ lập trình C

Ngôn ngữ lập trình C là một ngôn ngữ lập trình máy tính đa mục đích, thủ tục, và mệnh lệnh, được phát triển vào năm 1972 bởi Dennis M. Ritchie tại Phòng thí nghiệm Điện thoại Bell để phát triển hệ điều hành UNIX.

Năm 1978, Brian Kernighan và Dennis Ritchie đã công bố mô tả đầu tiên về ngôn ngữ C, hiện được biết đến như là tiêu chuẩn K&R.

C là ngôn ngữ máy tính được sử dụng rộng rãi nhất. Nó thường xuyên đứng đầu về mức độ phổ biến cùng với ngôn ngữ lập trình Java, cũng là một ngôn ngữ phổ biến và được sử dụng rộng rãi nhất trong số các lập trình viên phần mềm hiện đại.

Các phiên bản của 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 |

## Tại sao nên học lập trình C

Ngôn ngữ lập trình C là một điều **BẮT BUỘC** đối với học sinh, sinh viên và các chuyên gia đang làm việc nếu muốn trở thành một Kỹ sư Phần mềm xuất sắc. Một số lý do quan trọng vì sao bạn nên học lập trình C:

* C là một ngôn ngữ lập trình có cấu trúc, và bạn có thể sử dụng những kỹ năng học được từ C để thành thạo các ngôn ngữ lập trình khác.
* Có thể sử dụng chương trình C để viết mã hiệu quả và phát triển các dự án mạnh mẽ.
* C là một ngôn ngữ cấp thấp, cho phép bạn tương tác trực tiếp hơn với phần cứng và bộ nhớ của máy tính.

## Thông tin về ngôn ngữ C

C là ngôn ngữ lập trình hệ thống được sử dụng rộng rãi và phổ biến nhất. Phần lớn các phần mềm tiên tiến hiện nay đều được triển khai bằng ngôn ngữ C. Thông tin về ngôn ngữ C:

* C được phát minh để viết một hệ điều hành có tên là UNIX. Hệ điều hành UNIX hoàn toàn được viết bằng C.
* C là ngôn ngữ kế thừa của ngôn ngữ B, được giới thiệu vào khoảng đầu những năm 1970.
* Ngôn ngữ này đã được chuẩn hóa vào năm 1988 bởi Viện Tiêu chuẩn Quốc gia Hoa Kỳ (ANSI).

# CÁC TÍNH NĂNG CỦA NGÔN NGỮ LẬP TRÌNH C

Ngôn ngữ lập trình C là một trong những ngôn ngữ phổ biến nhất nhờ vào các tính năng mạnh mẽ và linh hoạt của nó. Dưới đây là các tính năng nổi bật của C:

## Ngôn ngữ thủ tục và có cấu trúc

C là một ngôn ngữ lập trình theo thủ tục (procedural), trong đó chương trình được xây dựng từ các bước tuần tự và hàm (function). Điều này giúp cho việc phát triển phần mềm trở nên có tổ chức và dễ dàng bảo trì.

## Ngôn ngữ đa mục đích

C không bị giới hạn trong một lĩnh vực ứng dụng cụ thể. Nó được sử dụng rộng rãi trong nhiều loại ứng dụng khác nhau, từ lập trình hệ thống như hệ điều hành, cơ sở dữ liệu, đến các ứng dụng phần mềm như chỉnh sửa ảnh.

## Ngôn ngữ lập trình nhanh

C là ngôn ngữ dựa trên trình biên dịch (compiler-based), giúp quá trình biên dịch và thực thi mã nguồn nhanh hơn. Mã nguồn được dịch trực tiếp sang mã máy, giúp CPU dễ dàng thực thi mà không cần máy ảo.

## Tính di động

Một chương trình C có thể chạy trên nhiều nền tảng máy tính khác nhau với rất ít hoặc không cần thay đổi mã nguồn, nhờ vào tính chất không phụ thuộc vào máy của ngôn ngữ này.

## Tính mở rộng

C cho phép mở rộng mã nguồn đã viết bằng cách thêm các tính năng mới hoặc chức năng mới mà không cần thay đổi cấu trúc cơ bản của chương trình. Điều này giúp chương trình dễ dàng được nâng cấp và bảo trì.

## Bộ thư viện chuẩn phong phú

Ngôn ngữ C đi kèm với một bộ thư viện chuẩn chứa nhiều hàm được tích hợp sẵn, như các hàm toán học, thao tác chuỗi, và các tiện ích của hệ điều hành. Ngoài ra, lập trình viên cũng có thể tạo ra các hàm tùy chỉnh và thêm chúng vào các thư viện này.

## Con trỏ

C cho phép sử dụng con trỏ, một trong những tính năng mạnh mẽ nhất của ngôn ngữ này. Con trỏ giúp thao tác trực tiếp với bộ nhớ và có thể được sử dụng để quản lý các thiết bị phần cứng, quản lý bộ nhớ động, và xử lý các ngắt.

## Ngôn ngữ cấp trung

C cung cấp sự kết hợp của các tính năng của ngôn ngữ cấp cao và quyền truy cập bộ nhớ ở mức thấp, làm cho nó trở thành một ngôn ngữ lập trình cấp trung. Điều này cho phép lập trình viên vừa có thể viết mã dễ đọc vừa có thể thao tác trực tiếp với phần cứng.

## Bộ toán tử phong phú

C có một bộ toán tử tích hợp phong phú, bao gồm các toán tử số học, logic, và các toán tử xử lý bit. Điều này cho phép viết các chương trình từ đơn giản đến phức tạp một cách linh hoạt.

## Đệ quy

C hỗ trợ đệ quy, nghĩa là một hàm có thể tự gọi chính nó để giải quyết các bài toán lặp lại hoặc phức tạp. Đệ quy giúp tái sử dụng mã và thực hiện các thuật toán phức tạp một cách dễ dàng.

## Kiểu dữ liệu do người dùng định nghĩa

C cho phép lập trình viên định nghĩa các kiểu dữ liệu tùy chỉnh như cấu trúc (struct), liên hiệp (union), và kiểu liệt kê (enum), mở rộng khả năng của các kiểu dữ liệu cơ bản.

## Chỉ thị tiền xử lý

C có các chỉ thị tiền xử lý như #include và #define, giúp quản lý các phần mã nguồn trước khi chúng được biên dịch, như nhập các thư viện hoặc định nghĩa các macro.

## Xử lý tệp tin

Mặc dù không phải là một phần của ngôn ngữ C cốt lõi, nhưng C cung cấp khả năng xử lý tệp tin thông qua các thư viện tiêu chuẩn, giúp thực hiện các thao tác đọc/ghi tệp tin một cách dễ dàng.

Nhờ các tính năng này, ngôn ngữ C đã trở thành một công cụ mạnh mẽ cho cả các lập trình viên mới bắt đầu và các chuyên gia phát triển phần mềm chuyên nghiệp.

# BUILD CHƯƠNG TRÌNH C/C++ BẰNG GCC

## Giới thiệu

Nhìn lại quá khứ, lúc mà các IDE còn chưa phát triển, lập trình viên phải thủ công build source code thông qua Terminal với các trình biên dịch C/C++ như gcc, g++. Thời đại công nghệ hiện nay có vô số IDE hỗ trợ build source code C/C++ dễ dàng.

|  |
| --- |
| **Câu hỏi:** Vậy tại sao ngày nay các IDE hỗ trợ mạnh mẽ như vậy mà phải học lại cách build source code C/C++ thủ công ?  **Trả lời:** Vẫn còn nhiều nền tảng không hỗ trợ các IDE như Server, không cần thiết phải có UI vì toàn bộ năng lực máy tính cần tập trung vào xử lý. |

## Kiểm tra hệ thống đã cài đặt trình biên dịch chưa?

Sử dụng lệnh version để kiểm tra phiên bản của trình biên dịch gcc hay g++ trong hệ điều hành Windows.

|  |
| --- |
| gcc –version  g++ –version |

Xem các tham số truyền vào khi sử dụng gcc hay g++ bằng lệnh.

|  |
| --- |
| gcc –help  g++ –help |

Hoặc đọc document về gcc hay g++.

|  |
| --- |
| man gcc  man g++ |

## Biên dịch mã nguồn C/C++ với gcc hay g++

Bắt đầu với 1 chương trình C đơn giản: **HelloWord**.

Tạo file **main.c** có nội dung như sau:

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* arg[]) {      printf("Hello World\n");      return 0;  } |

### Lệnh build chương trình C bằng GCC

Cú pháp:

|  |
| --- |
| gcc <source file> -o <outputfile> |

Cụ thể: **gcc main.c -o main**

Sau khi gọi lệnh này, gcc sẽ build ra file chương trình với tên là **main** và có khả năng thực thi. Để thực thi chương trình **main**, gõ lệnh như sau: **./main**

Kết quả xuất ra sẽ là: Hello World

Ngoài ra có thể sử dụng 1 trong các lệnh bên dưới.

**gcc -o <output file> <source file>**

hoặc

**gcc -Wall <source file> -o <outputfile>**

hoặc

**gcc -Wall -o <output file> <source file>**

* **-Wall**: sẽ in ra tất cả các warning trong quá trình build.
* **-o**: có nghĩa là build source file ra file thực thi.

### Lệnh build chương trình C bằng G++

Nếu build với g++ hoàn toàn tương tự chỉ thay thế gcc bằng g++.

Cụ thể: **g++ main.c -o main**

Tiến hành build lại để có file thực thi **main**. Và sử dụng lệnh**./main** để chạy chương trình.

# BUILD CHƯƠNG TRÌNH C/C++ BẰNG GCC - NÂNG CAO

C là một ngôn ngữ lập trình biên dịch. Các ngôn ngữ lập trình biên dịch cung cấp hiệu suất thực thi nhanh hơn so với các ngôn ngữ thông dịch. Có nhiều trình biên dịch khác nhau có thể được sử dụng để biên dịch một chương trình C, chẳng hạn như GCC, Clang, MSVC, v.v. Trong chương này, chúng ta sẽ giải thích những gì diễn ra ở hậu trường khi bạn biên dịch một chương trình C bằng trình biên dịch GCC.

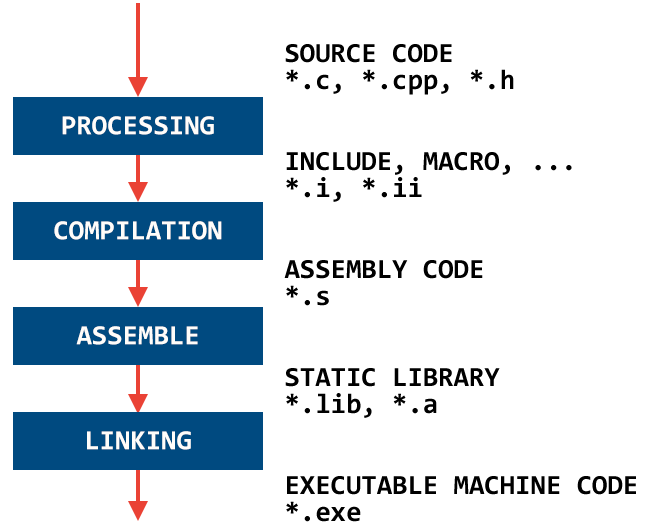
## Biên dịch một chương trình C

Một chuỗi các chỉ thị nhị phân bao gồm các bit 1 và 0 được gọi là mã máy. Các ngôn ngữ lập trình cấp cao như C, C++, Java, v.v. bao gồm các từ khóa gần gũi hơn với ngôn ngữ con người, chẳng hạn như tiếng Anh. Do đó, một chương trình viết bằng C (hoặc bất kỳ ngôn ngữ cấp cao nào khác) cần được chuyển đổi thành mã máy tương ứng. Quá trình này được gọi là biên dịch.

Lưu ý rằng mã máy phụ thuộc vào kiến trúc phần cứng và hệ điều hành. Nói cách khác, mã máy của một chương trình C được biên dịch trên một máy tính sử dụng hệ điều hành Windows sẽ không tương thích với một máy tính khác sử dụng hệ điều hành Linux. Vì vậy, chúng ta phải sử dụng trình biên dịch phù hợp với hệ điều hành mục tiêu.

## Các bước trong quá trình biên dịch C

Trong hướng dẫn này, chúng ta sẽ sử dụng gcc. Trình biên dịch gcc hỗ trợ nhiều ngôn ngữ lập trình khác nhau, bao gồm C. Để sử dụng nó, chúng ta cần cài đặt phiên bản tương thích với máy tính mục tiêu.



**Hình 1:** Quá trình biên dịch để build 1 chương trình

Quá trình build 1 chương trình trải qua 4 bước như dưới đây:

* **Bước 1**: Tiền xử lý (Preprocessing) - Bộ tiền xử lý thay thế giá trị của các directive như: #define, #include, #if, ... trong file mã nguồn, xóa bỏ các chú thích.
* **Bước 2**: Biên dịch (Compiling) - Trình biên dịch chuyển mã nguồn thành mã assembly.
* **Bước 3**: Dịch mã máy (Assembling) - Chuyển từ mã assembly sang mã nhị phân (mã máy).
* **Bước 4**: Liên kết (Linking)- Liên kết các module và các thư viện liên kết thành file thực thi.

## Build module C/C++ với GCC

Để hiểu rõ hơn về quá trình này, hãy cùng xem xét 2 ví dụ sau.

**VD1:** File main.c

|  |
| --- |
| #include <stdio.h>  int main(){       /\* my first program in C \*/       printf("Hello World! \n");     return 0;  } |

Để mã nguồn "main.c" có thể thực thi được, chúng ta cần nhập lệnh "gcc main.c", và quá trình biên dịch sẽ đi qua tất cả bốn bước mà nó bao gồm.

**Bước 1: Tiền xử lý (Preprocessing)**

Trình tiền xử lý thực hiện các hành động sau:

* Loại bỏ tất cả các nhận xét (comments) trong tệp nguồn.
* Bao gồm mã của các tệp tiêu đề (header files), tức là các tệp có phần mở rộng .h chứa các khai báo hàm C và các định nghĩa macro.
* Thay thế tất cả các macro (những đoạn mã đã được đặt tên) bằng giá trị của chúng.
* Đầu ra của bước này sẽ được lưu trữ trong một tệp với phần mở rộng .i, do đó tệp ở đây sẽ là "main.i".

Để dừng quá trình biên dịch ngay sau bước này, chúng ta có thể sử dụng tùy chọn "-E" với lệnh gcc trên tệp nguồn và nhấn Enter.

|  |
| --- |
| gcc -E main.c |

**Bước 2: Biên dịch (Compiling)**

Trình biên dịch tạo ra mã IR (Intermediate Representation - biểu diễn trung gian) từ tệp đã được tiền xử lý, do đó bước này sẽ tạo ra một tệp .s. Nói cách khác, các trình biên dịch khác có thể tạo ra mã lắp ráp (assembly code) ở bước này của quá trình biên dịch.

Chúng ta có thể dừng sau bước này với tùy chọn "-S" trên lệnh gcc, và nhấn Enter.

|  |
| --- |
| gcc -S main.c |

Đây là những gì tệp main.s sẽ trông như thế này:

|  |
| --- |
| .file   "helloworld.c"     .text     .def \_\_main; .scl    2;  .type   32; .endef     .section .rdata,"dr"  .LC0:     .ascii "Hello, World! \0"     .text     .globl   main     .def main;   .scl    2;  .type   32; .endef     .seh\_proc    main  main:     pushq    %rbp     .seh\_pushreg %rbp     movq %rsp, %rbp     .seh\_setframe    %rbp, 0     subq $32, %rsp     .seh\_stackalloc  32     .seh\_endprologue     call \_\_main     leaq .LC0(%rip), %rcx     call puts     movl $0, %eax     addq $32, %rsp     popq %rbp     ret     .seh\_endproc     .ident   "GCC: (x86\_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0"     .def puts;   .scl    2;  .type   32; .endef |

**Bước 3: Dịch mã máy (Assembling)**

Trình dịch mã (assembler) sẽ lấy mã IR và chuyển đổi nó thành mã đối tượng (object code), tức là mã ở ngôn ngữ máy (tức là nhị phân). Bước này sẽ tạo ra một tệp có phần mở rộng là .o.

Chúng ta có thể dừng quá trình biên dịch sau bước này bằng cách sử dụng tùy chọn "-c" với lệnh gcc, và nhấn Enter.

Lưu ý rằng tệp "main.o" không phải là một tệp văn bản, do đó nội dung của nó sẽ không đọc được khi bạn mở tệp này bằng trình soạn thảo văn bản.

**Bước 4: Liên kết (Linking)**

Trình liên kết (linker) tạo ra tệp thực thi cuối cùng, dưới dạng nhị phân. Nó liên kết mã đối tượng của tất cả các tệp nguồn lại với nhau. Trình liên kết biết nơi để tìm các định nghĩa hàm trong các thư viện tĩnh (static libraries) hoặc thư viện động (dynamic libraries).

* **Thư viện tĩnh (Static libraries):** Là kết quả của việc trình liên kết sao chép tất cả các hàm thư viện đã sử dụng vào tệp thực thi.
* **Thư viện động (Dynamic libraries):** Mã trong các thư viện động không được sao chép hoàn toàn, chỉ có tên của thư viện được đặt trong tệp nhị phân.

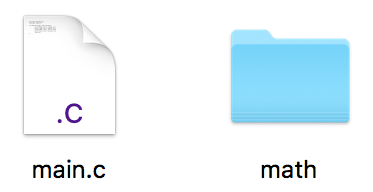
Theo mặc định, sau bước thứ tư và cũng là bước cuối cùng này, khi bạn gõ toàn bộ lệnh "gcc main.c" mà không có tùy chọn nào khác, trình biên dịch sẽ tạo ra một chương trình thực thi có tên là main.exe trong trường hợp dùng Windows.

Chúng ta cũng có thể chọn tạo ra một chương trình thực thi với tên mà chúng ta mong muốn bằng cách thêm tùy chọn "-o" vào lệnh gcc, đặt sau tên của tệp hoặc các tệp mà chúng ta đang biên dịch.

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

Bây giờ, bạn có thể gõ ./hello để thực thi mã đã biên dịch. Kết quả sẽ là "Hello World" và sau đó dấu nhắc lệnh của shell sẽ xuất hiện lại.

**VD 2:** Giả sử trong Project có 1 module **math**, chức năng tính toán cộng trừ nhân chia.



Trong thư mục **math** có 2 file là **math.h** định nghĩa prototype của các hàm và **math.c** hiện thực các prototype trong **math.h**.

**File math.h**

|  |
| --- |
| #ifndef \_\_MATH\_H\_\_  #define \_\_MATH\_H\_\_  int add(int a, int b);  int sub(int a, int b);  int mul(int a, int b);  int div(int a, int b);  #endif |

**File math.c**

|  |
| --- |
| #include "math.h"  int add(int a, int b) {      return a + b;  }  int sub(int a, int b) {      return a - b;  }  int mul(int a, int b) {      return a \* b;  }  int div(int a, int b) {      return a / b;  } |

**File main.c**

|  |
| --- |
| #include <stdio.h>  #include "math/math.h"  int main(int argc, char\* argv[]) {      printf("Add: %d\n", add(5, 6));      printf("Sub: %d\n", sub(10, 6));      printf("Mul: %d\n", mul(5, 40));      printf("Div: %d\n", div(60, 6));      return 0;  } |

Trong file **main.c** sử dụng module math, vì vậy trước khi build **main.c** phải build **math.c** sau đó link **math.o** (được build từ math.c).

* Bước 1: build module math
* gcc -c math/math.c
* Sau khi build sẽ được file **math.o**
* Bước 2: link **math.o** và **main.c**
* gcc -Wall -o main math.o main.c
* Sau khi chạy xong sẽ tạo được file thực thi file **main** bằng lệnh: **./main**

**Kết quả:**

|  |
| --- |
| Add: 11  Sub: 4  Mul: 200  Div: 10 |

## Link thư viện với GCC

### Thư viện là gì?

Trong lập trình C, thư viện là một tập hợp các hàm và mã mà bạn có thể sử dụng trong chương trình của mình. Thư viện giúp tái sử dụng mã và giảm công sức viết lại các hàm phổ biến.

Có hai loại thư viện chính mà quá trình linking có thể sử dụng:

**1. Thư viện tĩnh (Static Libraries):**

* Các thư viện này có phần mở rộng .a trên hệ thống UNIX/Linux hoặc .lib trên Windows.
* Khi liên kết với thư viện tĩnh, tất cả mã từ thư viện được sao chép vào tệp thực thi cuối cùng.
* Điều này làm cho tệp thực thi lớn hơn, nhưng nó độc lập và không cần phụ thuộc vào thư viện ngoài khi chạy.

**2. Thư viện động (Shared Libraries):**

* Các thư viện này có phần mở rộng .so trên hệ thống UNIX/Linux hoặc .dll trên Windows.
* Khi liên kết với thư viện động, mã từ thư viện không được sao chép vào tệp thực thi. Thay vào đó, chỉ có tên của thư viện được lưu trong tệp thực thi.
* Khi chạy chương trình, hệ điều hành sẽ tải thư viện này vào bộ nhớ.
* Điều này giúp tiết kiệm không gian bộ nhớ và cho phép cập nhật thư viện mà không cần biên dịch lại chương trình.

### Linking là gì

**Linking** là bước cuối cùng trong quá trình biên dịch chương trình C. Nó có nhiệm vụ kết hợp mã nguồn của bạn với các thư viện và các đoạn mã khác để tạo ra một tệp thực thi duy nhất.

Khi bạn viết mã trong C, bạn thường sử dụng các hàm từ thư viện chuẩn hoặc thư viện do người khác tạo ra. Những thư viện này không phải lúc nào cũng được biên dịch cùng với mã của bạn mà được lưu trữ riêng biệt. Quá trình linking sẽ kết nối mã nguồn của bạn với các thư viện đó để tạo ra chương trình hoàn chỉnh.

### Quá Trình Linking Hoạt Động Như Thế Nào

Giả sử bạn viết một chương trình đơn giản main.c như sau:

|  |
| --- |
| #include <stdio.h>  int main() {      printf("Hello, World!\n");      return 0;  } |

Khi bạn biên dịch chương trình này bằng lệnh gcc main.c -o main, quá trình linking sẽ diễn ra như sau:

1. **Preprocessing:** Loại bỏ nhận xét, mở rộng macro, và bao gồm các tệp tiêu đề (.h).
2. **Compilation:** Chuyển mã nguồn từ C thành mã máy ở dạng mã trung gian.
3. **Assembling:** Chuyển mã trung gian thành mã máy, lưu trong các tệp đối tượng (.o).
4. **Linking:** Kết hợp các tệp đối tượng với các thư viện cần thiết để tạo ra tệp thực thi.

Trong trường hợp này, hàm printf được định nghĩa trong thư viện chuẩn C (libc). Trình liên kết sẽ tìm kiếm thư viện này và liên kết nó với mã nguồn của bạn để tạo ra tệp thực thi hoàn chỉnh. Nếu thiếu thư viện cần thiết, quá trình linking sẽ thất bại.

### Cách liên kết thư viện bằng lệnh gcc

Khi sử dụng gcc để biên dịch và liên kết chương trình với một thư viện, bạn sử dụng các tùy chọn -I và -L:

**1. Tùy chọn -I**

Dùng để chỉ ra thư viện mà bạn muốn liên kết.

Cú pháp:

|  |
| --- |
| -I<library\_name> |

**VD**: Nếu bạn muốn liên kết với thư viện toán học (libm), bạn sẽ sử dụng -lm. Trình liên kết sẽ tìm kiếm libm.a (thư viện tĩnh) hoặc libm.so (thư viện động) và liên kết nó với chương trình của bạn.

2. **Tùy chọn -L**

Dùng để chỉ ra thư mục mà trình liên kết cần tìm kiếm các thư viện.

Cú pháp:

|  |
| --- |
| -L<directory> |

**VD**: Nếu thư viện của bạn nằm trong thư mục /usr/local/lib, bạn cần sử dụng -L/usr/local/lib để đảm bảo trình liên kết có thể tìm thấy nó.

### Ví dụ quá trình Linking chi tiết

Giả sử bạn có tệp nguồn main.c:

|  |
| --- |
| #include <stdio.h>  #include "mylib.h"  // Thư viện tự tạo  int main() {      mylib\_function();  // Gọi hàm từ thư viện mylib      return 0;  } |

Và bạn có một thư viện tĩnh tên là mylib.a chứa hàm mylib\_function():

**1. Biên dịch và liên kết**

|  |
| --- |
| gcc main.c -L/usr/local/lib -lmylib -o main |

**2. Giải thích**

* **-L/usr/local/lib**: Nói với trình liên kết tìm thư viện trong thư mục /usr/local/lib.
* **-lmylib**: Liên kết với thư viện libmylib.a hoặc libmylib.so trong thư mục đó.
* **-o main**: Tạo ra tệp thực thi tên là main.

**3. Kết quả**

* Trình biên dịch sẽ tạo ra một tệp thực thi main, trong đó hàm mylib\_function() đã được liên kết từ thư viện mylib.a.

# TOKEN TRONG NGÔN NGỮ C

Ngôn ngữ lập trình C là một trong những ngôn ngữ cơ bản và được sử dụng rộng rãi trong lập trình. Để hiểu về C, bạn cần nắm vững các khái niệm cơ bản như token (thành phần nhỏ nhất trong mã nguồn), các loại token khác nhau và vai trò của chúng trong quá trình biên dịch. Sau đây là các khái niệm được giải thích một cách đơn giản và logic:

## Token trong ngôn ngữ C

**Token** là đơn vị nhỏ nhất trong mã nguồn của một ngôn ngữ lập trình như C. Bạn có thể hình dung token giống như từ vựng trong ngôn ngữ tự nhiên (như tiếng Anh), nơi mà một câu được tạo thành từ các từ, số, và ký hiệu chấm câu. Trong lập trình C, trình biên dịch sẽ phân tách mã nguồn thành các token này trước khi tiếp tục với các giai đoạn biên dịch tiếp theo.

## Các loại Token trong C

Mã nguồn trong C được tạo thành từ nhiều loại token khác nhau. Các loại token bao gồm:

### ****Bộ ký tự (Character set)****:

Ngôn ngữ lập trình C sử dụng bộ ký tự ASCII (American Standard Code for Informations Interchange). Theo chuẩn này, bộ kí tự gồm có 256 kí tự đó là:

* Các chữ cái: A … Z, a .. z
* Các chữ số: 0 1 2 3 4 5 6 7 8 9
* Các dấu phép toán số học: +,-,\*,/,...
* Các ký tự đặc biệt trong lập trình C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Các ký tự đặc biệt trong lập trình C** | | | | |
| , | < | > | . | ~ |
| ( | ) | ; | $ | : |
| % | [ | ] | # | ? |
| ' | & | { | } | " |
| ^ | ! | \* | / | | |

Mỗi kí tự có tương ứng 1 số duy nhất gọi là mã, trong đó có 128 kí tự đầu (có mã từ 0 tới 127) là kí tự cố định và 128 kí tự còn lại (có mã từ 128 tới 255) là các kí tự mở rộng, tức là nó có thể thay đổi tuỳ theo ngôn ngữ mỗi quốc gia sử dụng.

### ****Từ khóa (Keywords)****

* + Từ khóa là các từ có sẵn trong ngôn ngữ C với ý nghĩa đặc biệt và không thể dùng cho mục đích khác. Ban đầu, C có 32 từ khóa, sau đó được bổ sung thêm trong các phiên bản chuẩn mới của C.
  + Ví dụ: int, return, if, else.

### ****Hằng (Literals)****

* + Hằng là giá trị cố định được gán trực tiếp cho biến trong mã nguồn. Hằng số có thể là số, ký tự, chuỗi, hoặc các dạng khác.
  + Ví dụ: 42 là một hằng số nguyên, "Hello" là một hằng chuỗi.

### **Định danh (Identifiers)**

* + Định danh là các tên do người lập trình đặt cho các biến, hàm, hoặc kiểu dữ liệu. Định danh phải tuân theo các quy tắc nhất định, ví dụ không được trùng với từ khóa.
  + Ví dụ: age, calculateSum.

1. **Toán tử (Operators)**
   * Toán tử là các ký hiệu đặc biệt dùng để thực hiện các phép tính hoặc so sánh trong C.
   * Ví dụ: + (cộng), - (trừ), \* (nhân), / (chia), < (bé hơn), > (lớn hơn).
2. **Ký hiệu đặc biệt (Special symbols)**:
   * Ngoài các toán tử, các ký hiệu đặc biệt khác như dấu phẩy, chấm phẩy, dấu ngoặc đơn cũng có vai trò quan trọng trong việc cấu trúc mã nguồn.
   * Ví dụ: { và } dùng để nhóm các câu lệnh thành một khối mã.

# CẤU TRÚC CHƯƠNG TRÌNH

Trong ngôn ngữ lập trình C, một chương trình cơ bản được cấu trúc một cách có tổ chức với các thành phần quan trọng như hàm main(), các thư viện tiêu chuẩn, và các câu lệnh. Dưới đây là một cái nhìn tổng quan về cấu trúc của một chương trình C.

## Chương trình 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:

**File hello.c**

|  |
| --- |
| #include <stdio.h>  // Thư viện chuẩn cho nhập/xuất  // Hàm chính của chương trình  int main(int argc, char\* argv[]) {      int a, b, sum;      // Gán giá trị cho các biến      a = 10;      b = 20;      // Tính tổng      sum = a + b;      // Hiển thị kết quả      printf("Tổng là: %d\n", sum);      return 0;  // Kết thúc chương trình  } |

### Thư viện Tiêu chuẩn

Mọi chương trình C thường bắt đầu với việc bao gồm các thư viện tiêu chuẩn cần thiết bằng cách sử dụng các chỉ thị tiền xử lý #include. Các thư viện này cung cấp các hàm và macro cần thiết cho chương trình.

|  |
| --- |
| # 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().**

### Hàm main()

Hàm main() là điểm bắt đầu của mọi chương trình C. Khi chương trình bắt đầu thực thi, CPU sẽ chạy mã nguồn trong hàm main().

Cấu trúc cơ bản của hàm main()

|  |
| --- |
| #include <stdio.h>  int main (int argc, char\* argv[]) {      // Các câu lệnh của chương trình      return 0;  } |

Trong đó:

* int: Kiểu dữ liệu trả về của hàm main(), thường là int để trả về giá trị nguyên cho hệ điều hành khi chương trình kết thúc.
* return 0;: Câu lệnh này báo hiệu rằng chương trình đã chạy thành công.

Các tham số của hàm main :

1. **int argc** : Đây là số lượng các tham số được truyền vào chương trình khi nó được chạy. argc viết tắt của “argument count” (số lượng tham số). Nó là một số nguyên (integer) và luôn luôn ít nhất bằng 1, vì chương trình luôn tính là một tham số (tên chương trình).
2. **char \*argv** : Đây là một mảng chứa các chuỗi (mảng ký tự) đại diện cho các tham số được truyền vào chương trình. argv viết tắt của “argument vector” (vector tham số). Điều quan trọng cần lưu ý là argv là một mảng hai chiều (một mảng các con trỏ đến mảng ký tự). Mỗi phần tử trong mảng argv là một con trỏ đến một chuỗi (cũng là một mảng ký tự).

### Khối ngoặc

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

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.

### Các Khai báo Biến

Trong chương trình C, các biến cần được khai báo trước khi sử dụng. Các biến này có thể được khai báo trong hàm main() hoặc bên ngoài hàm để dùng chung trong toàn bộ chương trình (biến toàn cục).

Ví dụ:

|  |
| --- |
| int a, b;  // Khai báo hai biến nguyên a và b |

### Các Câu lệnh

Chương trình C bao gồm các câu lệnh (statements) thực hiện các thao tác cụ thể như tính toán, hiển thị, hoặc kiểm soát luồng chương trình. Mỗi câu lệnh kết thúc bằng dấu chấm phẩy (;).

Ví dụ:

|  |
| --- |
| a = 10;  b = 20;  int sum = a + b;  printf("Tổng là: %d\n", sum); |

### Các Hàm

Một chương trình C có thể có nhiều hàm để thực hiện các nhiệm vụ cụ thể. Hàm main() là hàm chính, nhưng bạn có thể tạo thêm các hàm khác và gọi chúng trong main() hoặc trong các hàm khác.

Ví dụ:

|  |
| --- |
| int tong(int x, int y) {      return x + y;  } |

### Nhận xét (Comments)

Nhận xét trong C không được biên dịch mà chỉ dành cho người đọc để giải thích mã. Nhận xét được đặt giữa /\* ... \*/ hoặc sau //.

Ví dụ:

|  |
| --- |
| // Đây là một nhận xét đơn dòng  /\* Đây là   \* nhận xét nhiều dòng  \*/ |

## 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 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.**

### Thực hiện chương trình

Sau khi file được biên dịch, mở **Terminal** để thực thi chương trình bằng cách nhập:

|  |
| --- |
| ./hello |

Khi thực thi, chương trình đã biên dịch sẽ in **Hello World**, theo sau là một dòng mới, tới dấu nhắc lệnh.

## Chú ý

|  |
| --- |
| **Câu hỏi:** Chi tiết về lệnh return 0 trong hàm main?  **Trả lời:**  **1. Chương Trình C và Hệ Điều Hành**   * Khi bạn chạy một chương trình C, hệ điều hành sẽ bắt đầu thực thi chương trình đó. * Sau khi chương trình chạy xong, nó sẽ trả về một giá trị cho hệ điều hành được gọi là **exit status** (trạng thái thoát). Giá trị này giúp hệ điều hành biết được chương trình đã hoàn thành một cách bình thường hay đã xảy ra lỗi trong quá trình chạy. * Giá trị này được lập trình viên quyết định, thường là bằng cách sử dụng lệnh return trong hàm main().   **2. Quy Ước Trong Lập Trình C**   * Theo quy ước chung trong lập trình C, giá trị 0 thường được hiểu là chương trình đã chạy thành công (Success), còn bất kỳ giá trị nào khác 0 (ví dụ 1, -1, 2, ...) được hiểu là chương trình gặp lỗi (Failure). * Đây không phải là quy tắc bắt buộc, mà là quy ước mà đa số lập trình viên tuân theo để đảm bảo tính nhất quán và dễ hiểu khi chương trình kết thúc.   **3. Hệ Điều Hành Dựa Vào Quy Ước Này Như Thế Nào?**   * Hệ điều hành (và nhiều công cụ khác như scripts, batch files, hoặc các chương trình khác) sử dụng giá trị trả về này để quyết định liệu chương trình đã chạy thành công hay gặp lỗi. * Quy ước 0 = Thành công, khác 0 = Lỗi được hiểu rộng rãi, vì vậy hệ điều hành sẽ dựa vào giá trị này để xử lý. Ví dụ:   + Nếu chương trình trả về 0, hệ điều hành ghi nhận rằng chương trình đã hoàn thành thành công.   + Nếu chương trình trả về giá trị khác 0, hệ điều hành có thể ghi nhận rằng đã có lỗi xảy ra và có thể thực hiện các bước tiếp theo như ghi log, cảnh báo người dùng, hoặc thực hiện các hành động sửa lỗi khác. |

# 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 "nhận xét"\*/  /\* Một lượng lớn mã với nhận xét nhiều dòng \*/  int foo() {      /\* rất nhiều mã \*/      ...        /\* ... một số nhận xét mô tả câu lệnh if ... \*/      if (someTest) {          /\* thêm một số nhận xét khác \*/          return 1;      }      return 0;  }  #endif /\* 0 \*/  /\* mã từ đây trở đi là "không bị nhận xét"\*/  ... |

## 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 \*/      }  } |

**Lưu ý:**

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.

**VD:** 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***

Phiên bản 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, nằm trên một dòng  } |

## Rủi ro có thể xảy ra do Trigraph

Trong lập trình C, Trigraph là một chuỗi ba ký tự đặc biệt được sử dụng để thay thế cho một ký tự đặc biệt khác mà không phải là ký tự trong bảng mã ASCII chuẩn.

Trigraph được sử dụng để tạo ra các ký tự đặc biệt như dấu ngoặc vuông, dấu nháy kép, dấu gạch ngược, v.v. trên các hệ thống hoặc thiết bị đầu cuối không hỗ trợ những ký tự đó.

Một số **Trigraph** phổ biến trong C:

| **Trigraph** | **Ký tự thay thế** |
| --- | --- |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | ` |
| ??< | { |
| ??> | } |
| ??- | ~ |

**VD:** Trigraph ??= → # Gây Vấn Đề Trong Chuỗi.

|  |
| --- |
| #include <stdio.h>  int main() {      char\* str = "Đây là một trigraph ??= sẽ trở thành dấu #";      printf("%s\n", str);      return 0;  } |

**Kết quả**

|  |
| --- |
| Đây là một trigraph # sẽ trở thành dấu # |

**Giải thích:** Trigraph ??= trong chuỗi "Đây là một trigraph ??= sẽ trở thành dấu #" sẽ được thay thế bởi ký tự #.

**VD:** Lỗi Trigraph gây ??/ thành ký tự nối dòng (\)

|  |
| --- |
| 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; |

**Giải thích:** Trigraph **??/** sẽ tạo thành một dấu gạch 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 sẽ gây ra lỗi chương trình vì biến bar chưa được khai báo.

# KIỂU DỮ LIỆU

## A diagram of data types Description automatically generatedKiểu dữ liệu

Kiểu dữ liệu của một biến còn thể hiện dung lượng không gian mà nó chiếm trong bộ nhớ và cách diễn giải mẫu bit được lưu trữ.

## Phân loại kiểu dữ liệu

Có 3 loại kiểu dữ liệu khác nhau trong ngôn ngữ C.

|  |  |  |
| --- | --- | --- |
| **STT** | **Loại kiểu dữ liệu** | **Mô tả** |
| 1 | Loại nguyên thủy | Số nguyên (char, int), số thực (float, double), ký tự (char) |
| 2 | Loại dẫn xuất |  |
| 3 | Loại do ngườ dung định nghĩa |  |

### Kiểu dữ liệu nguyên thủy

Ngôn ngữ C có 4 kiểu dữ liệu cơ bản là: char, interger, float, double

|  |  |
| --- | --- |
| **Kiểu dữ liệu cơ bản** | |
| **Tên** | **Kích thước (Byte)** |
| char | 1 |
| int | 4 |
| float | 4 |
| double | 8 |

Kích thước trong bộ nhớ và miền giá trị của các kiểu dữ liệu còn phụ thuộc vào hệ thống và chương trình dịch tương ứng. Giá trị được đưa ra ở đây là trên hệ thống Windows 64 bit và trình dịch GCC MinGW.

##### Kiểu số nguyên

Trong ngôn ngữ C, có 4 bổ ngữ để sửa đổi kiểu dữ liệu. Các bổ ngữ này kết hợp với các kiểu dữ liệu chính để phân loại phạm vi sử dụng Tối ưu bộ nhớ lưu trữ.

|  |  |  |
| --- | --- | --- |
| **Các bổ ngữ** | **Mục đích** | **Mô tả** |
| signed | Thay đổi khoảng giá trị của byte. | Có dấu |
| unsigned | Không dấu |
| long | Thay đổi giá trị byte. | 4 byte |
| short | 2 byte |

**Bảng:** Giá trị kiểu dữ liệu số nguyên.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tên** | **Mô tả** | **Kích thước** | **Khoảng giá trị** |
| char | Số nguyên bé  (Không nên dùng để lưu số nguyên) | 1 byte | signed: -  unsigned: 0 |
| short | Số nguyên ngắn | 2 byte | signed: -  unsigned: 0 |
| int | Số nguyên | 4 byte | signed: -  unsigned: 0 |
| long | Số nguyên dài | 4 byte | signed: -  unsigned: 0 |
| long long | Số nguyên cực dài | 8 byte | signed: -  unsigned: 0 |

Ngoài ra, một số kiểu dữ liệu khác như:

|  |  |  |
| --- | --- | --- |
| **Tên** | **Kích thước** | **Khoảng giá trị** |
| uint8\_t | 1 byte | 0 |
| uint16\_t | 2 byte | 0 |
| uint32\_t | 4 byte | 0 |

##### Kiểu số thực

Kiểu số thực được dùng để chứa những số có dấu phẩy động.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tên** | **Kích thước** | **Khoảng giá trị** | **Độ chính xác** |
| float | 4 | 1.2E-38 🡪 3.4E+38 | Độ chính xác 6 con số |
| double | 8 | 2.3E-308 🡪 1.7E+308 | Độ chính xác 15 con số |
| long double | 16 | 3.4E-4932 🡪 1.1E+4932 | Độ chính xác 19 con số |

E = 10 mũ

##### Kiểu ký tự

Kiểu ký tự dùng để chứa những ký tự.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tên** | **Mô tả** | **Kích thước** | **Khoảng giá trị** |
| char | Kí tự | 1 byte | - |
| unsigned char | 0 |

#### Kiểu bool

Giá trị 1: là đúng (true)

Giá trị 0: là sai (false)

Trong C quy định sai là 0 hoặc **null** (\0)

#### Kiểu void

Kiểu void có nghĩa là “không có giá trị”, nó không được dùng trong khai báo biến thông  
thường mà được sử dụng để chỉ định kiểu của các hàm không có giá trị trả về. Xác định kích thước kiểu dữ liệu

#### Toán tử sizeof là gì?

Toán tử sizeof nhận một tham số là bất kỳ kiểu dữ liệu nào và trả về kích thước của kiểu dữ liệu đó.

**Cú pháp:**

|  |
| --- |
| sizeof (Kiểu dữ liệu) |

#### Kiểm tra kích thước các kiểu dữ liệu

|  |
| --- |
| **VD:** Xác định kích thước của các kiểu dữ liệu cơ bản trong C. |
| #include<stdio.h>  int main() {      printf("Kich thuoc kieu char: %i\n", sizeof(char));      printf("Kich thuoc kieu int: %i\n", sizeof(int));      printf("Kich thuoc kieu float: %i\n", sizeof(float));      printf("Kich thuoc kieu double: %i\n", sizeof(double));      return 0;  } |

**Kết quả:**

|  |
| --- |
| Kich thuoc kieu char: 1  Kich thuoc kieu int: 4  Kich thuoc kieu float: 4  Kich thuoc kieu double: 8 |

**VD:** Xác định kích thước của các kiểu dữ liệu **– Mở rộng thêm**

|  |
| --- |
| //Xác định kích thước của các kiểu dữ liệu  #include <stdio.h>      int main() {      char c;      short s;      int i;      unsigned int ui;      float f;      double d;      long l;      long long ll;      unsigned long long ull;      // Lấy kích thước từng kiểu dữ liệu//      printf("Size cua kieu char: %i\n",sizeof(c));      printf("Size cua kieu short: %i\n",sizeof(s));      printf("Size cua kieu int: %i\n",sizeof (i));      printf("Size cua kieu unsigned int: %i\n",sizeof (ui));      printf("Size cua kieu float: %i\n",sizeof (f));      printf("Size cua kieu double: %i\n",sizeof (d));      printf("Size cua kieu long: %i\n",sizeof (l));      printf("Size cua kieu long long: %i\n",sizeof (ll));      printf("Size cua kieu unsigned long long: %i\n",sizeof (ull));      return 0;  } |

**Kết quả:**

|  |
| --- |
| Size cua kieu char: 1  Size cua kieu short: 2  Size cua kieu int: 4  Size cua kieu unsigned int: 4  Size cua kieu float: 4  Size cua kieu double: 8  Size cua kieu long: 4  Size cua kieu long long: 8  Size cua kieu unsigned long long: 8 |

**VD:** Xác định kích thước max, min của kiểu dữ liệu số nguyên

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #include <float.h>  int main() {      printf("\n------------Signed -------------\n");      printf("char      - Bit: %d\n", CHAR\_BIT);      printf("char      - max: %d - min: %d\n",CHAR\_MAX, CHAR\_MIN);      printf("short     - max: %d - min: %d\n",SHRT\_MAX, SHRT\_MIN);      printf("int       - max: %d - min: %d\n",INT\_MAX, INT\_MIN);      printf("long      - max: %ld - min: %ld\n",LONG\_MAX, LONG\_MIN);      printf("long long - max: %lld - min: %lld\n",LLONG\_MAX, LLONG\_MIN);      printf("\n------------Unsigned -------------\n");      printf("unsigned char      - max: %u\n",UCHAR\_MAX);      printf("unsigned short     - max: %u\n",USHRT\_MAX);      printf("unsigned int       - max: %u\n",UINT\_MAX);      printf("unsigned long      - max: %lu\n",ULONG\_MAX);      printf("unsigned long long - max: %llu\n",ULLONG\_MAX);      return 0;  } |

**Kết quả:**

|  |
| --- |
| ------------Signed -------------  char      - Bit: 8  char      - max: 127 - min: -128  short     - max: 32767 - min: -32768  int       - max: 2147483647 - min: -2147483648  long      - max: 2147483647 - min: -2147483648  long long - max: 9223372036854775807 - min: -9223372036854775808  ------------Unsigned -------------  unsigned char      - max: 255  unsigned short     - max: 65535  unsigned int       - max: 4294967295  unsigned long      - max: 4294967295  unsigned long long - max: 18446744073709551615 |

Tệp tiêu đề float.h xác định các macro cho phép bạn sử dụng các giá trị này và các chi tiết khác về biểu diễn nhị phân của số thực trong chương trình của bạn. Ví dụ sau sẽ in không gian lưu trữ được thực hiện bởi một kiểu float và các giá trị phạm vi của nó:

**VD:** Xác định kích thước max, min của kiểu dữ liệu số thực

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #include <float.h>  int main() {      printf("float       - max: %-20g - min: %-20g\n",FLT\_MAX, FLT\_MIN);      printf("double      - max: %-20g - min: %-20g\n",DBL\_MAX, DBL\_MIN);      printf("long double - max: %-20g - min: %-20g\n",LDBL\_MAX, LDBL\_MIN);      return 0;  } |

**Kết quả:**

|  |
| --- |
| float - max: 3.40282e+038 - min: 1.17549e-038  double - max: 1.79769e+308 - min: 2.22507e-308  long double - max: 3.20528e-317 - min: 3.20528e-317 |

# BIẾN SỐ

## Khái niệm

Trong lập trình, **biến** (variable) là một khái niệm rất quan trọng và phổ biến. Đơn giản, biến là một **vị trí trong bộ nhớ** của máy tính (RAM) được sử dụng để **lưu trữ dữ liệu** mà chương trình có thể thay đổi trong quá trình thực thi.

## Đặc điểm đặc điểm của biến trong C

* **Tên**: Mỗi biến có một tên (định danh) để phân biệt với các biến khác. Tên biến giúp bạn tham chiếu đến vùng nhớ đó trong mã nguồn.
* **Kiểu dữ liệu**: Mỗi biến được gán với một kiểu dữ liệu, chẳng hạn như số nguyên (int), số thực (float), ký tự (char), v.v. Kiểu dữ liệu xác định loại dữ liệu mà biến có thể lưu trữ và các thao tác có thể thực hiện trên nó.
* **Giá trị**: Giá trị của biến là dữ liệu cụ thể mà biến đang lưu trữ tại một thời điểm nào đó. Giá trị này có thể thay đổi trong quá trình chương trình thực thi.
* **Địa chỉ bộ nhớ**: Mỗi biến có một địa chỉ bộ nhớ duy nhất, là nơi dữ liệu của biến được lưu trữ.
* **Phạm vi và vòng đời của biến**:
* **Phạm vi (scope)** của biến đề cập đến khu vực trong mã nguồn nơi biến có thể được truy cập.
  + Biến cục bộ: Khai báo bên trong một hàm và chỉ có thể truy cập trong hàm đó.
  + Biến toàn cục: Khai báo bên ngoài tất cả các hàm và có thể truy cập từ bất kỳ hàm nào trong cùng tệp mã nguồn.
* **Vòng đời (lifetime)** của biến chỉ thời gian mà biến tồn tại trong bộ nhớ:
  + Biến cục bộ tồn tại trong thời gian hàm thực thi.
  + Biến toàn cục tồn tại trong suốt chương trình.
* **Biến tĩnh (static variables)**:Là biến mà giá trị của nó được bảo toàn giữa các lần gọi hàm và được khai báo bằng từ khóa static.

## Quy tắc đặt tên biến

Biến số là một định danh nên sẽ tuân theo quy tắc đặt định danh.

### Khai báo biến

Trước khi sử dụng một biến, bạn phải khai báo nó. Việc khai báo cho phép trình biên dịch biết được kiểu dữ liệu của biến và lượng bộ nhớ cần cấp phát cho nó.

**Cú pháp:**

|  |
| --- |
| <Kiểu dữ liệu> <Danh sách tên biến>; |

Trong đó

* Kiểu dữ liệu: Là các kiểu dữ liệu cơ bản, dẫn xuất, bool, …
* Danh sách tên biến: gồm các tên biến có cùng kiểu dữ liệu, mỗi tên biến cách nhau dấu phẩy.

**VD:** Khai báo biến trong C

|  |
| --- |
| int age;  float pi;  char initial; |

### Khởi tạo biến

Khi khai báo biến, bạn có thể đồng thời gán giá trị ban đầu cho nó, gọi là khởi tạo biến.

**VD:** Khởi tạo biến. (Khai báo trước, gán giá trị sau)

|  |
| --- |
| int a, b, c;  a = 1;  b = 2;  c = 5; |

**Ví dụ:** Khởi tạo biến (Vừa khai báo vừa gán giá trị)

|  |
| --- |
| int a = 1, b = 2, c = 5;  float wight = 10.5, height = 20.5; |

### Phạm vi của biến

**Phạm vi của biến** (variable scope) là khái niệm mô tả khu vực trong mã nguồn nơi biến đó có thể được truy cập hoặc sử dụng. Trong ngôn ngữ C, có ba phạm vi chính của biến:

**Lưu ý**: Khi lập trình, bạn phải nắm rõ phạm vi của biến. Nếu khai báo và sử dụng không đúng, không rõ ràng sẽ dẫn đến sai sót khó kiểm soát được, vì vậy bạn cần phải xác định đúng vị trí, phạm vi sử dụng biến trước khi sử dụng biến.

#### ****Phạm vi cục bộ (Local Scope)****

* Biến cục bộ là biến được khai báo bên trong một khối mã (thường là bên trong một hàm hoặc một cặp dấu {}).
* Biến cục bộ chỉ có thể được truy cập trong khối mã nơi nó được khai báo.
* Sau khi khối mã kết thúc, biến cục bộ sẽ bị hủy và bộ nhớ được giải phóng.

**VD:** Biến cục bộ

|  |
| --- |
| #include<stdio.h>  void myFunction() {      int x = 10; // Biến cục bộ x      printf("%d\n", x); // Có thể sử dụng x trong hàm này  }  int main() {      myFunction();      // printf("%d\n", x); // Lỗi: không thể truy cập x ở đây      return 0;  } |

Trong ví dụ này, biến x chỉ tồn tại và có thể truy cập trong hàm myFunction. Nếu cố gắng sử dụng x ngoài hàm này, sẽ xảy ra lỗi biên dịch.

#### ****Phạm vi toàn cục (Global Scope)****

* Biến toàn cục là biến được khai báo bên ngoài tất cả các hàm, thường là ở đầu tệp mã nguồn.
* Biến toàn cục có thể được truy cập từ bất kỳ hàm nào trong cùng tệp mã nguồn.
* Biến toàn cục tồn tại trong suốt thời gian chương trình chạy.

**VD:** Biến toàn cục

|  |
| --- |
| #include<stdio.h>  int y = 20; // Biến toàn cục y  void myFunction() {      printf("%d\n", y); // Có thể truy cập y ở đây  }  int main() {      printf("%d\n", y); // Có thể truy cập y ở đây      return 0;  } |

Trong ví dụ này, biến y được khai báo toàn cục, vì vậy nó có thể được truy cập cả trong hàm myFunction và main.

#### ****Phạm vi khối (Block Scope)****

* Phạm vi khối là một trường hợp đặc biệt của phạm vi cục bộ, khi biến được khai báo bên trong một cặp dấu {} nhưng không nằm trong hàm.
* Biến chỉ tồn tại trong khối {} đó và không thể truy cập bên ngoài khối.

**VD:** Biến toàn cục

|  |
| --- |
| #include<stdio.h>  int main() {      int z = 30;      {          int z = 40; // Biến z mới với phạm vi khối          printf("%d\n", z); // In ra 40      }      printf("%d\n", z); // In ra 30      return 0;  } |

Trong ví dụ này, có hai biến z, một biến z toàn cục trong hàm main và một biến z cục bộ trong khối {}. Biến z bên trong khối {} chỉ tồn tại trong khối đó, và giá trị của nó không ảnh hưởng đến biến z bên ngoài khối.

# BIẾN TĨNH

Biến tĩnh (static variable) trong C là một khái niệm quan trọng và phức tạp, liên quan đến cách quản lý bộ nhớ, phạm vi truy cập và thời gian tồn tại của biến trong chương trình. Dưới đây là một trình bày chi tiết về biến tĩnh trong C:

## ****Định nghĩa về biến tĩnh****

**Biến tĩnh** là biến có từ khóa static trong khai báo. Từ khóa này thay đổi cách biến được quản lý bởi trình biên dịch về thời gian tồn tại và phạm vi của nó.

## ****Phạm vi và thời gian tồn tại****

Biến tĩnh có hai đặc điểm chính:

* **Thời gian tồn tại (Lifetime):** Biến tĩnh được khởi tạo chỉ một lần và tồn tại trong suốt thời gian chạy của chương trình, ngay cả khi nó ra khỏi phạm vi của khối lệnh.
* **Phạm vi (Scope):** Biến tĩnh có thể có phạm vi cục bộ hoặc toàn cục, nhưng trong mọi trường hợp, phạm vi của nó bị hạn chế để tránh bị truy cập ngoài mong muốn.

## ****Biến tĩnh cục bộ****

Biến tĩnh cục bộ được khai báo bên trong một hàm. Phạm vi của nó chỉ nằm trong hàm đó, nhưng thời gian tồn tại thì kéo dài suốt chương trình.

**Cách khai báo:**

|  |
| --- |
| void function() {      static int localStaticVar = 10;      // Code here  } |

**Đặc điểm:**

* + Khi hàm được gọi nhiều lần, biến tĩnh cục bộ không bị khởi tạo lại. Giá trị của nó được giữ nguyên từ lần gọi trước.
  + Giá trị mặc định của biến tĩnh cục bộ nếu không được khởi tạo là 0.

**VD:**

|  |
| --- |
| #include<stdio.h>  void countCalls() {      static int count = 0;      count++;      printf("Called %d times\n", count);  }  int main() {      countCalls();  // Output: Called 1 times      countCalls();  // Output: Called 2 times      return 0;  } |

Ở đây, biến count sẽ giữ giá trị giữa các lần gọi hàm countCalls.

## ****Biến Tĩnh Toàn Cục****

Biến tĩnh toàn cục được khai báo bên ngoài mọi hàm, thông thường ở đầu file mã nguồn.

**Cách khai báo**

|  |
| --- |
| static int globalStaticVar = 20; |

**Đặc điểm**

* Phạm vi của biến tĩnh toàn cục chỉ nằm trong file nơi nó được khai báo, khác với biến toàn cục thông thường có thể được truy cập từ các file khác (bằng cách sử dụng từ khóa extern).
* Điều này giúp giảm nguy cơ xung đột tên biến khi chương trình lớn có nhiều file mã nguồn.

**VD:**

|  |
| --- |
| #include<stdio.h>  static int count = 0;  void increment() {      count++;  }  void printCount() {      printf("Count: %d\n", count);  }  int main() {      increment();      printCount();  // Output: Count: 1      return 0;  } |

Biến count trong ví dụ trên không thể truy cập từ các file khác ngoài file nơi nó được khai báo.

## ****Biến Tĩnh Trong Hàm Main và Các Khối Lệnh Khác****

Biến tĩnh có thể được khai báo trong hàm main hoặc các khối lệnh khác. Nguyên tắc hoạt động của nó không thay đổi: giá trị của biến tĩnh sẽ tồn tại cho đến khi chương trình kết thúc.

## ****Khởi tạo Biến Tĩnh****

* Biến tĩnh có thể được khởi tạo với một giá trị cụ thể hoặc để mặc định (giá trị mặc định là 0 đối với các kiểu số và NULL đối với con trỏ).
* Nếu biến tĩnh không được khởi tạo rõ ràng, trình biên dịch sẽ tự động khởi tạo nó với giá trị 0.

## ****Lợi ích và Hạn chế của Biến Tĩnh****

**Lợi ích:**

* **Lưu giữ trạng thái:** Biến tĩnh cục bộ có thể lưu giữ trạng thái giữa các lần gọi hàm mà không cần sử dụng biến toàn cục, giúp cải thiện tính đóng gói và bảo mật mã nguồn.
* **Giảm thiểu xung đột tên:** Biến tĩnh toàn cục giúp hạn chế xung đột tên giữa các file khác nhau trong một dự án lớn.
* **Hiệu quả bộ nhớ:** Do biến tĩnh được khởi tạo chỉ một lần và tồn tại suốt chương trình, nó giúp tiết kiệm tài nguyên hệ thống khi cần lưu trữ thông tin liên tục.

**Hạn chế:**

* **Khó quản lý:** Sử dụng quá nhiều biến tĩnh có thể làm phức tạp mã nguồn và khó khăn trong việc quản lý trạng thái chương trình, đặc biệt trong các dự án lớn.
* **Khó khăn trong kiểm thử:** Do trạng thái của biến tĩnh được giữ lại giữa các lần gọi hàm, việc kiểm thử các hàm sử dụng biến tĩnh có thể khó khăn hơn, vì cần phải kiểm soát trạng thái của các biến này một cách cẩn thận.

## ****Một số Lưu ý Khi Sử Dụng Biến Tĩnh****

* **Không lạm dụng:** Sử dụng biến tĩnh quá mức có thể dẫn đến các lỗi khó phát hiện và giảm tính trong sáng của mã nguồn.
* **Cẩn thận với khởi tạo:** Nếu khởi tạo biến tĩnh với các biểu thức phức tạp, cần đảm bảo rằng kết quả luôn mong muốn vì biến chỉ được khởi tạo một lần duy nhất.
* **Sử dụng hợp lý:** Sử dụng biến tĩnh khi thực sự cần thiết, chẳng hạn như khi cần lưu giữ trạng thái giữa các lần gọi hàm hoặc cần giới hạn phạm vi của một biến toàn cục trong một file cụ thể.

## ****So sánh Biến Tĩnh và Biến Toàn Cục****

**Phạm vi:**

* Biến toàn cục: Có thể truy cập từ bất kỳ đâu trong chương trình nếu được khai báo extern.
* Biến tĩnh toàn cục: Chỉ có thể truy cập trong file khai báo.

**Thời gian tồn tại:**

* Cả hai đều có thời gian tồn tại suốt chương trình.

### Biến static (Biến tĩnh)

##### Biến static trong khai báo biến cục bộ

Khi 1 biến cục bộ được khai báo với từ khóa static. Biến sẽ chỉ được khởi tạo 1 lần duy nhất và tồn tại suốt thời gian chạy chương trình. Giá trị của nó không bị mất đi ngay cả khi kết thúc hàm. Mỗi lần hàm được gọi, giá trị của biến chính bằng giá trị tại lần gần nhất hàm được gọi.

**VD.**

|  |
| --- |
| #include<stdio.h>  int in\_so\_thu\_tu(void){      static int x = 0;      x = x + 1;      printf("%d\r\n",x);  }  int main() {      in\_so\_thu\_tu ();      in\_so\_thu\_tu ();      in\_so\_thu\_tu ();      in\_so\_thu\_tu ();      in\_so\_thu\_tu ();   return 0;  } |

**Kết quả:**

|  |
| --- |
| **1**  **2**  **3**  **4**  **5** |

##### Biến static trong khai báo biến toàn cục và khai báo hàm

Mỗi project thường sẽ được viết trên nhiều File vì mục đích phân chia module cũng như là để dễ bảo trì. Do có nhiều File nên rất có thể ở các File sẽ có sự trùng lặp trong cách đặt tên biến. Để tránh sự cố sai sót này người ta đưa ra khái niệm biến toàn cục tĩnh và hàm tĩnh.

* Biến toàn cục tĩnh sẽ chỉ có thể được truy cập và sử dụng trong File khai báo nó, các File khác không có cách nào truy cập được.
* Hàm tĩnh sẽ chỉ có thể gọi trong File khai báo nó, các File khác không có cách nào gọi hàm này được.

**VD.**

|  |
| --- |
| //-----------------  //A.c  // biến a này chỉ được sử dụng trong file A.c  static int a;  // hàm hienthi() này chỉ được sử dụng trong file A.c  static void hien\_thi() {};  int c;  //------------------  //B.c  // biến a này chỉ được sử dụng trong file B.c  static int a;  // hàm hienthi() này chỉ được sử dụng trong file B.c  static void hien\_thi() {};  int d; |

## 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 \* " tham chiếu " thể hiện một con trỏ.
* Toán tử binary [] " phần tử mảng " thể hiện một mảng.
* Toán tử (1+n)-ary () " 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 |
| \* 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.

|  |
| --- |
|  |

## 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"; |

# DEFINE

Trong ngôn ngữ lập trình C, #define là một trong những chỉ thị tiền xử lý (preprocessor directive) được sử dụng phổ biến. Nó cung cấp một cách để tạo ra các hằng số, macro, và các định nghĩa thay thế trong mã nguồn. Dưới đây là một cái nhìn chi tiết về #define và các ứng dụng của nó trong C.

## ****Định nghĩa về**** #define

**#define** là một chỉ thị tiền xử lý trong C được sử dụng để định nghĩa các hằng số hoặc các macro. Khi chương trình được biên dịch, trình tiền xử lý sẽ thay thế tất cả các chỗ xuất hiện của từ hoặc biểu thức được định nghĩa bởi #define bằng giá trị hoặc mã tương ứng trước khi quá trình biên dịch thực sự bắt đầu.

**Cú pháp:**

|  |
| --- |
| #define NAME replacement\_text |

Trong đó:

* + NAME là tên của hằng số hoặc macro.
  + replacement\_text là giá trị hoặc đoạn mã sẽ thay thế cho NAME khi trình tiền xử lý thực hiện.

## ****Hằng Số Bằng**** #define

* #define thường được sử dụng để tạo ra các hằng số mà không sử dụng từ khóa const.
* **Ví dụ:**
* **Ví dụ:**

c

Sao chép mã

#define PI 3.14

#define MAX\_BUFFER\_SIZE 1024

Trong ví dụ trên, bất cứ chỗ nào trong mã nguồn có PI sẽ được thay thế bằng 3.14, và MAX\_BUFFER\_SIZE sẽ được thay thế bằng 1024 trong quá trình tiền xử lý.

## ****Macro trong**** #define

* Macro là một tính năng của #define cho phép bạn định nghĩa các đoạn mã có thể nhận tham số đầu vào, tương tự như hàm, nhưng hoạt động ở cấp độ tiền xử lý.
* **Cú pháp:**

c

Sao chép mã

#define MACRO\_NAME(parameters) replacement\_text

* **Ví dụ:**

c

Sao chép mã

#define SQUARE(x) ((x) \* (x))

Trong ví dụ này, bất kỳ chỗ nào có SQUARE(x) trong mã nguồn sẽ được thay thế bằng ((x) \* (x)). Khi bạn gọi SQUARE(5), nó sẽ được thay thế thành ((5) \* (5)).

* **Lưu ý:**
  + Để đảm bảo các phép toán trong macro được thực hiện đúng thứ tự, bạn nên đặt các tham số trong ngoặc đơn như trong ví dụ trên.
  + Macro không kiểm tra kiểu dữ liệu, vì vậy cần cẩn thận khi sử dụng để tránh các lỗi không mong muốn.

## #define ****và Điều Kiện Tiền Xử Lý****

* #define thường được sử dụng cùng với các chỉ thị điều kiện như #if, #ifdef, #ifndef, #else, và #endif để tạo ra mã nguồn điều kiện.
* **Ví dụ:**

c

Sao chép mã

#define DEBUG 1

#ifdef DEBUG

printf("Debugging mode is on.\n");

#endif

Ở đây, nếu DEBUG được định nghĩa (tức là khác 0), đoạn mã in ra thông báo "Debugging mode is on." sẽ được biên dịch.

## ****Các Kỹ Thuật và Ứng Dụng Của**** #define

* **Tạo Hằng Số:**
  + #define có thể được sử dụng để định nghĩa các hằng số phổ biến mà không cần sử dụng bộ nhớ thêm.

c

Sao chép mã

#define TRUE 1

#define FALSE 0

* **Tạo Macro Nâng Cao:**
  + Bạn có thể sử dụng #define để tạo các macro phức tạp, chẳng hạn như các kiểm tra lỗi, tính toán toán học, hoặc các cấu trúc điều khiển.

c

Sao chép mã

#define MAX(a,b) ((a) > (b) ? (a) : (b))

Macro này sẽ trả về giá trị lớn hơn giữa a và b.

* **Tối Ưu Hóa Code:**
  + Sử dụng macro để tránh lặp lại mã, cải thiện tính dễ đọc và bảo trì.

c

Sao chép mã

#define PRINT\_INT(x) printf("Value of " #x " is %d\n", x)

Macro này sử dụng tính năng "stringize" (#x) để biến tham số thành một chuỗi ký tự, sau đó in ra cùng với giá trị của nó.

* **Loại Bỏ Mã Không Cần Thiết:**
  + Trong quá trình phát triển phần mềm, bạn có thể cần phải tạm thời loại bỏ một số đoạn mã không cần thiết bằng cách sử dụng #define cùng với #ifdef hoặc #ifndef.

c

Sao chép mã

#define FEATURE\_ENABLED

#ifdef FEATURE\_ENABLED

// Code for the feature

#endif

Khi FEATURE\_ENABLED được định nghĩa, đoạn mã trong #ifdef sẽ được biên dịch; nếu không, đoạn mã sẽ bị bỏ qua.

## ****Lưu Ý Khi Sử Dụng**** #define

* **Không có Kiểm Tra Kiểu Dữ Liệu:** #define chỉ thay thế văn bản, không kiểm tra kiểu dữ liệu. Do đó, sử dụng #define để thay thế hằng số hoặc tạo macro phức tạp có thể dẫn đến lỗi nếu không cẩn thận.
* **Gây Khó Hiểu Cho Mã Nguồn:** Việc sử dụng quá nhiều #define, đặc biệt là với các macro phức tạp, có thể làm mã nguồn trở nên khó hiểu và khó bảo trì.
* **Không Nên Sử Dụng #define Thay Cho Hàm:** Trong hầu hết các trường hợp, sử dụng hàm thông thường là lựa chọn tốt hơn, vì hàm có thể kiểm tra kiểu dữ liệu và giúp mã nguồn dễ hiểu hơn.

## ****Một Số Ứng Dụng Khác Của**** #define

* **Định Nghĩa Chỉ Một Lần:**
  + #define thường được sử dụng để đảm bảo một file tiêu đề (header file) chỉ được nạp một lần vào trình biên dịch, tránh lỗi "redefinition".

c

Sao chép mã

#ifndef HEADER\_FILE\_NAME\_H

#define HEADER\_FILE\_NAME\_H

// Nội dung file header

#endif

* **Định Nghĩa Đường Dẫn Tùy Chỉnh:**
  + Định nghĩa các đường dẫn tùy chỉnh hoặc các thông số môi trường để dễ dàng thay đổi mà không cần sửa mã nguồn.

c

Sao chép mã

#define LOG\_FILE\_PATH "/var/log/mylogfile.log"

## ****Kết Luận****

#define là một công cụ mạnh mẽ trong C, cung cấp một cách linh hoạt để định nghĩa hằng số và macro. Khi sử dụng đúng cách, nó có thể làm cho mã nguồn dễ đọc hơn, dễ bảo trì hơn và có thể tối ưu hóa hiệu suất. Tuy nhiên, cần cẩn thận để tránh các lỗi không mong muốn do việc thay thế văn bản mà không có kiểm tra kiểu hoặc lỗi trong mã nguồn. Trong nhiều trường hợp, việc sử dụng const và hàm có thể là lựa chọn tốt hơn so với #define, đặc biệt là trong các dự án lớn hoặc phức tạp.

#### Số

Sau đây là một ví dụ về cách bạn sử dụng chỉ thị #define để xác định một hằng số:

|  |
| --- |
| #define AGE 10 |

Trong ví dụ này, hằng số có tên AGE sẽ chứa giá trị là 10

#### Chuỗi

Bạn có thể sử dụng chỉ thị #define để định nghĩa một hằng chuỗi.

Ví dụ:

|  |
| --- |
| #define NAME "TechOnTheNet.com" |

Trong ví dụ này, hằng số được gọi là NAME sẽ chứa giá trị của "TechOnTheNet.com".

Dưới đây là một chương trình C ví dụ nơi chúng tôi xác định hai hằng số này:

|  |
| --- |
| #include <stdio.h>  #define NAME "TechOnTheNet.com"  #define AGE 10  int main()  {  printf("%s is over %d years old.\n", NAME, AGE);  return 0;  } |

Chương trình C này sẽ in như sau:

|  |
| --- |
| TechOnTheNet.com is over 10 years old. |

#### Biểu thức

Bạn có thể sử dụng lệnh #define để xác định một hằng số bằng một biểu thức.

Ví dụ:

|  |
| --- |
| #define AGE (20 / 2) |

Trong ví dụ này, hằng số có tên AGE cũng sẽ chứa giá trị là 10.

Dưới đây là một chương trình C ví dụ trong đó chúng tôi sử dụng một biểu thức để xác định hằng số:

|  |
| --- |
| #include <stdio.h>  #define AGE (20 / 2)  int main()  {  printf("TechOnTheNet.com is over %d years old.\n", AGE);  return 0;  } |

Chương trình C này cũng sẽ in như sau:

|  |
| --- |
| TechOnTheNet.com is over 10 years old. |

## Câu hỏi phỏng vấn

### ****Khái niệm cơ bản về**** #define

* **Câu hỏi:** #define là gì? Nó hoạt động như thế nào trong C?
* **Gợi ý trả lời:** #define là một chỉ thị tiền xử lý (preprocessor directive) trong C, được sử dụng để định nghĩa các hằng số hoặc các macro. Trình tiền xử lý sẽ thay thế tất cả các chỗ xuất hiện của từ hoặc biểu thức được định nghĩa bằng giá trị hoặc mã tương ứng trước khi quá trình biên dịch thực sự bắt đầu.

### 2. ****So sánh**** #define ****và**** const

* **Câu hỏi:** Bạn có thể so sánh #define và const trong C không? Khi nào thì nên dùng #define và khi nào nên dùng const?
* **Gợi ý trả lời:** #define là chỉ thị tiền xử lý, thực hiện thay thế văn bản trước khi biên dịch, không có kiểm tra kiểu dữ liệu. const là từ khóa trong C, cho phép khai báo biến với giá trị không thay đổi, có sự kiểm tra kiểu dữ liệu và có thể gỡ lỗi dễ dàng. Sử dụng const khi cần kiểm tra kiểu và muốn có khả năng bảo trì và gỡ lỗi tốt hơn, còn #define nên dùng cho các macro hoặc điều kiện tiền xử lý.

Trong C, cả #define và const đều được sử dụng để tạo ra các hằng số, nhưng chúng hoạt động khác nhau và có các đặc điểm riêng biệt. Dưới đây là sự phân biệt chi tiết giữa #define và const:

1. **Cơ Chế Hoạt Động**

* **#define:**
  + Là một chỉ thị tiền xử lý (preprocessor directive), không phải là một lệnh của ngôn ngữ C.
  + Trình tiền xử lý sẽ thay thế mọi chỗ xuất hiện của tên được định nghĩa bằng giá trị hoặc biểu thức thay thế trong quá trình tiền xử lý, trước khi mã được biên dịch.
  + Không có kiểm tra kiểu dữ liệu khi sử dụng #define, chỉ là sự thay thế văn bản trực tiếp.
* **const:**
  + Là một từ khóa trong ngôn ngữ C, dùng để khai báo một biến có giá trị không thay đổi sau khi được khởi tạo.
  + const biến được trình biên dịch quản lý giống như các biến thông thường nhưng không thể thay đổi giá trị của nó sau khi được khởi tạo.
  + Có sự kiểm tra kiểu dữ liệu tại thời điểm biên dịch.

2. **Cú Pháp và Ví Dụ**

* **#define:**

c

Sao chép mã

#define PI 3.14

#define MAX\_BUFFER\_SIZE 1024

* + Bất cứ chỗ nào trong mã nguồn có PI sẽ được thay thế bằng 3.14, và MAX\_BUFFER\_SIZE sẽ được thay thế bằng 1024 trong quá trình tiền xử lý.
* **const:**

c

Sao chép mã

const float pi = 3.14;

const int maxBufferSize = 1024;

* + pi là một biến số thực có giá trị 3.14 và không thể thay đổi giá trị này sau khi được khởi tạo.

3. **Phạm Vi (Scope)**

* **#define:**
  + #define có phạm vi áp dụng trong toàn bộ file mã nguồn sau khi nó được khai báo.
  + Không có phạm vi cụ thể như các biến thông thường, nên dễ gây xung đột tên nếu không quản lý cẩn thận, đặc biệt là trong các dự án lớn hoặc khi sử dụng trong nhiều file.
* **const:**
  + const tuân theo các quy tắc phạm vi (scope) của biến trong C. Có thể là phạm vi cục bộ (local) hoặc toàn cục (global), tùy thuộc vào nơi khai báo.
  + Có thể khai báo nhiều biến const cùng tên trong các phạm vi khác nhau mà không bị xung đột.

4. **Kiểm Tra Kiểu Dữ Liệu (Type Checking)**

* **#define:**
  + Không có kiểm tra kiểu dữ liệu vì #define chỉ đơn thuần là sự thay thế văn bản. Điều này có thể dẫn đến các lỗi khó phát hiện, chẳng hạn như khi sử dụng macro phức tạp.

Ví dụ:

c

Sao chép mã

#define SQUARE(x) ((x) \* (x))

int a = SQUARE(3 + 2); // Kết quả sẽ là ((3 + 2) \* (3 + 2)) = 25 chứ không phải 5 \* 5 = 25

* **const:**
  + Trình biên dịch sẽ kiểm tra kiểu dữ liệu của các biến const, giúp ngăn chặn các lỗi liên quan đến kiểu.

Ví dụ:

c

Sao chép mã

const int max = 100;

int x = max + 10; // Kiểu dữ liệu hợp lệ, không có lỗi

5. **Sử Dụng Bộ Nhớ**

* **#define:**
  + Không sử dụng bộ nhớ cho các giá trị được định nghĩa vì nó chỉ là sự thay thế văn bản trước khi biên dịch.
  + Điều này có thể làm cho mã nguồn lớn hơn sau khi biên dịch, đặc biệt là nếu giá trị được định nghĩa xuất hiện nhiều lần trong chương trình.
* **const:**
  + const là một biến thực sự trong bộ nhớ, và do đó nó chiếm một vị trí bộ nhớ trong thời gian chạy.
  + Tuy nhiên, việc sử dụng const có thể tiết kiệm bộ nhớ hơn nếu giá trị hằng được sử dụng nhiều lần, vì trình biên dịch có thể tối ưu hóa bộ nhớ cho nó.

6. **Gỡ Lỗi và Bảo Trì**

* **#define:**
  + Gỡ lỗi có thể khó khăn hơn vì khi biên dịch, mã nguồn đã bị thay thế bởi trình tiền xử lý, do đó bạn không thể thấy các định nghĩa #define trong các công cụ gỡ lỗi như gdb.
  + Khi bảo trì mã nguồn, các lỗi có thể phát sinh do sự thay thế văn bản không mong muốn.
* **const:**
  + Gỡ lỗi dễ dàng hơn vì các biến const tồn tại trong mã biên dịch và có thể được kiểm tra trực tiếp trong quá trình gỡ lỗi.
  + Bảo trì mã nguồn tốt hơn, đặc biệt là trong các dự án lớn, vì kiểu dữ liệu được kiểm tra và các lỗi liên quan đến kiểu được ngăn chặn từ sớm.

7. **Các Ứng Dụng và Khuyến Nghị**

* **#define:**
  + Nên sử dụng để định nghĩa các hằng số đơn giản hoặc các macro mà không yêu cầu kiểm tra kiểu dữ liệu.
  + Hữu ích cho việc định nghĩa các macro phức tạp hoặc điều kiện tiền xử lý (như cấu hình môi trường hoặc loại bỏ mã không cần thiết).
* **const:**
  + Nên sử dụng cho các hằng số mà cần kiểm tra kiểu hoặc khi cần sử dụng chúng như các biến thông thường trong mã nguồn.
  + Tăng cường tính bảo mật và rõ ràng của mã nguồn, đồng thời cải thiện khả năng bảo trì.

8. **Ví dụ So Sánh**

* **Sử dụng #define:**

c

Sao chép mã

#define LENGTH 10

#define WIDTH 5

#define AREA (LENGTH \* WIDTH)

* **Sử dụng const:**

c

Sao chép mã

const int length = 10;

const int width = 5;

const int area = length \* width;

Trong ví dụ này, cả hai đều tạo ra kết quả tương tự, nhưng const cung cấp sự an toàn và kiểm tra kiểu dữ liệu, trong khi #define chỉ đơn giản là thay thế văn bản.

9. **Kết Luận**

* **#define** là công cụ mạnh mẽ cho việc thay thế văn bản, macro đơn giản, và điều kiện tiền xử lý, nhưng thiếu sự kiểm tra kiểu dữ liệu và có thể gây ra lỗi khó phát hiện.
* **const** cung cấp sự an toàn về kiểu dữ liệu, dễ bảo trì, và khả năng gỡ lỗi tốt hơn, nhưng có thể chiếm thêm bộ nhớ.

Trong thực tế, khi cần định nghĩa hằng số đơn giản, const thường là lựa chọn tốt hơn vì nó tăng tính an toàn và dễ bảo trì hơn. Tuy nhiên, #define vẫn có vị trí của nó trong việc tạo macro và điều kiện tiền xử lý.

### ****Lợi ích và hạn chế của**** #define

* **Câu hỏi:** Lợi ích và hạn chế của việc sử dụng #define là gì?
* **Gợi ý trả lời:**
  + **Lợi ích:** Dễ sử dụng, không tốn bộ nhớ vì chỉ là sự thay thế văn bản, có thể định nghĩa macro phức tạp, hỗ trợ điều kiện tiền xử lý.
  + **Hạn chế:** Không có kiểm tra kiểu dữ liệu, có thể gây ra lỗi khó phát hiện, không hỗ trợ gỡ lỗi dễ dàng, có thể dẫn đến xung đột tên và làm giảm tính dễ đọc của mã nguồn.

### ****Macro trong**** #define

* **Câu hỏi:** Bạn có thể định nghĩa một macro sử dụng #define để tính diện tích hình chữ nhật với chiều dài và chiều rộng cho trước không?
* **Gợi ý trả lời:**

c

Sao chép mã

#define AREA(length, width) ((length) \* (width))

### ****Các vấn đề với**** #define

* **Câu hỏi:** Bạn có thể gặp phải vấn đề gì khi sử dụng macro với #define? Làm sao để giảm thiểu những vấn đề đó?
* **Gợi ý trả lời:**
  + Vấn đề chính là thiếu kiểm tra kiểu dữ liệu và sự thay thế không mong muốn, có thể dẫn đến các lỗi logic. Để giảm thiểu, nên đặt các tham số của macro trong dấu ngoặc đơn và cần hiểu rõ ngữ cảnh sử dụng. Sử dụng các hàm thay thế cho macro khi cần thiết để có sự kiểm tra kiểu và bảo trì tốt hơn.

### ****Ví dụ về lỗi với**** #define

* **Câu hỏi:** Hãy giải thích tại sao đoạn mã sau đây cho kết quả không mong đợi:

c

Sao chép mã

#define SQUARE(x) x \* x

int result = SQUARE(3 + 2);

* **Gợi ý trả lời:** Do cách #define chỉ thay thế văn bản, nên SQUARE(3 + 2) sẽ thành 3 + 2 \* 3 + 2, theo thứ tự ưu tiên toán tử trong C, kết quả là 3 + 6 + 2 = 11 thay vì 25. Để tránh lỗi này, nên viết #define SQUARE(x) ((x) \* (x)).

### ****Điều kiện tiền xử lý với**** #define

* **Câu hỏi:** Bạn có thể sử dụng #define để tạo điều kiện tiền xử lý trong C không? Cho ví dụ.
* **Gợi ý trả lời:**

c

Sao chép mã

#define DEBUG

#ifdef DEBUG

printf("Debug mode is active.\n");

#endif

### ****Chuyển đổi macro thành hàm****

* **Câu hỏi:** Khi nào bạn nên chuyển một macro #define thành một hàm trong C?
* **Gợi ý trả lời:** Khi cần kiểm tra kiểu dữ liệu, gỡ lỗi dễ dàng hơn, và khi macro quá phức tạp hoặc dễ gây ra lỗi logic. Hàm cung cấp tính an toàn và khả năng bảo trì tốt hơn so với macro.

### #define ****trong bảo trì mã nguồn****

* **Câu hỏi:** Tại sao việc sử dụng quá nhiều #define có thể làm cho việc bảo trì mã nguồn trở nên khó khăn?
* **Gợi ý trả lời:** Sử dụng quá nhiều #define có thể dẫn đến việc mã nguồn trở nên khó hiểu do sự thay thế văn bản không rõ ràng, thiếu kiểm tra kiểu dữ liệu, và có thể gây ra lỗi mà không dễ dàng phát hiện trong quá trình gỡ lỗi.

### #define ****và phạm vi (scope)****

* **Câu hỏi:** Phạm vi của một #define là gì? Có cách nào để giới hạn phạm vi của một #define không?
* **Gợi ý trả lời:** #define có phạm vi từ vị trí nó được định nghĩa cho đến cuối file hoặc đoạn mã được bao gồm. Không thể giới hạn phạm vi trực tiếp, nhưng có thể sử dụng các chỉ thị tiền xử lý như #undef để ngừng hiệu lực của một #define.

# 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.

## 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).

**Lưu ý:** 1 ở đây là True và 0 ở đây là False.

|  |  |
| --- | --- |
| **Toán tử** | **Ý nghĩa** |
| == | So sánh bằng |
| > | So sánh lớn hơn |
| < | So sánh bé hơn |
| != | So sánh khác |
| >= | Lớn hơn hoặc bằng |
| <= | Bé hơn hoặc bằng |

### Toán tử ==

**VD**: Toán tử “=”. Kiểm tra xem các toán hạng được cung cấp có bằng nhau không.

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

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

### Toán tử !=

Kiểm tra xem toán hạng bên trái có giá trị khác toán hạng bên phải hay không.

**VD**: Toán tử “!=”. Kiểm tra xem các toán hạng được cung cấp có bằng nhau không.

|  |
| --- |
|  |

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

### Toán tử !

**VD**: Toán tử “!”. Kiểm tra xem các toán hạng được cung cấp có bằng nhau không.

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

|  |
| --- |
| !someVal |

Điều này có tác dụng tương tự như

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

### Toán tử >

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

**VD:** Toán tử “>”.

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

### Toán tử <

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

**VD:** Toán tử “<”.

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

### Toán tử >=

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

**VD:** Toán tử “>=”.

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

### Toán tử <=

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

**VD:** Toán tử “<=”.

|  |
| --- |
| 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. 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(int argc, char\* argv[]) {  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 |

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

|  |
| --- |
| #include <stdio.h>  int main(int argc, char\* argv[]) {  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>  int main(int argc, char\* argv[]) {  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.

VD:

|  |
| --- |
| 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

Toán tử số học trả về một giá trị là kết quả của việc áp dụng toán hạng bên trái cho toán hạng bên phải. Toán tử số học gồm các phép toán: cộng, trừ, nhân, chia lấy nguyên, chia lấy dư.

|  |  |
| --- | --- |
| **Toán tử** | **Ý nghĩa của toán tử** |
| + | Phép cộng |
| - | Phép trừ |
| \* | Phép nhân |
| / | Phép chia lấy phần nguyên |
| % | Phép chia lấy dư (Chỉ áp dụng cho số nguyên) |

***Thứ tự toán tử số học***

|  |  |
| --- | --- |
| %, /, \* | +, - |

### Toán tử cộng

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

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

**Kết quả**

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

### Toán tử trừ

**VD:** Toán tử trừ (-) được sử dụng để trừ toán hạng thứ 2 cho toán hạng thứ nhất.

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

**Kết quả**

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

### Toán tử nhân

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

|  |
| --- |
| #include<stdio.h>  int main(int argc, char\* argv[]) {  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 (\*)**

### 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 argc, char\* argv[]) {  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 argc, char\* argv[]) {  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 |

### 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(int argc, char\* argv[]) {  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;  } |

### 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(int argc, char\* argv[]) {  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).

### 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". \*/ |

### 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(int argc, char\* argv[]) {  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". \*/  } |

### 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 argc, char\* argv[]) {  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 |

### 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". \*/ |

### 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". \*/ |

### 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". \*/ |

## Toán tử Sizeof

Toán tử sizeof trả về kích thước dữ liệu của đối tượng đó.

Cú pháp:

|  |
| --- |
| sizeof (Kiểu dữ liệu) |

### 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, trả về 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! \*/ |

### 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

### 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. \*/ |

### 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. \*/ |

### 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

### Cộng con trỏ

### Trừ con trỏ

## \_Alignof

# BOOLEAN

## 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(int argc, char\* argv[]) {  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.

## 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(int argc, char\* argv[]) {  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>.

## 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(int argc, char\* argv[]) {  \_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.

## 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.

## 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