Lập trình

**C++**

# TỔNG QUAN

C++ là ngôn ngữ lập trình dạng tự do, được biên dịch, có mục đích chung, phân biệt chữ hoa chữ thường, được nhập tĩnh, hỗ trợ lập trình thủ tục, hướng đối tượng và chung.

C ++ được coi là ngôn ngữ cấp trung bình, vì nó bao gồm sự kết hợp của cả các tính năng của ngôn ngữ cấp cao và cấp thấp.

C++ được phát triển bởi Bjarne Stroustrup bắt đầu từ năm 1979 tại Bell Labs ở Murray Hill, New Jersey, như một cải tiến cho ngôn ngữ C và ban đầu được đặt tên là C with Classes nhưng sau đó nó được đổi tên thành C++ vào năm 1983.

C++ là siêu bộ của C, và hầu như mọi chương trình C hợp pháp đều là chương trình C++ hợp pháp.

Lưu ý - Một ngôn ngữ lập trình được cho là sử dụng kiểu gõ tĩnh khi kiểm tra kiểu được thực hiện trong thời gian biên dịch trái ngược với thời gian chạy.

## **Object-Oriented Programming**

C++ hỗ trợ đầy đủ lập trình hướng đối tượng bao gồm 4 đặc tính:

* Trừu tượng
* Đóng gói
* Kế thừa
* Đa hình

## **Thư viện tiêu chuẩn**

C++ tiêu chuẩn bao gồm 3 thư phần quan trọng:

* Ngôn ngữ cốt lõi cung cấp tất cả các khối xây dựng bao gồm biến, kiểu dữ liệu và chữ, v.v.
* Thư viện chuẩn C++ cung cấp một tập hợp phong phú các hàm thao tác với tệp, chuỗi, v.v.
* Thư viện mẫu chuẩn (STL) cung cấp một tập hợp phong phú các phương thức thao tác với cấu trúc dữ liệu, v.v.

## **Tiêu chuẩn ANSI**

Tiêu chuẩn ANSI là một nỗ lực để đảm bảo rằng C++ có thể mang theo được; mã bạn viết cho trình biên dịch của Microsoft sẽ biên dịch không có lỗi, sử dụng trình biên dịch trên máy Mac, UNIX, hộp Windows hoặc Alpha.

Tiêu chuẩn ANSI đã ổn định trong một thời gian và tất cả các nhà sản xuất trình biên dịch C++ chính đều hỗ trợ tiêu chuẩn ANSI.

## **Khi học C++**

Điều quan trọng nhất khi học C++ là tập trung vào các khái niệm.

Mục đích của việc học một ngôn ngữ lập trình là để trở thành một lập trình viên giỏi hơn; nghĩa là trở nên hiệu quả hơn trong việc thiết kế và triển khai các hệ thống mới cũng như duy trì các hệ thống cũ.

C++ hỗ trợ nhiều phong cách lập trình khác nhau. Bạn có thể viết theo phong cách Fortran, C, Smalltalk, v.v., bằng bất kỳ ngôn ngữ nào. Mỗi phong cách có thể đạt được mục tiêu của nó một cách hiệu quả trong khi duy trì thời gian chạy và hiệu quả không gian.

## **Sử dụng C++**

C++ được sử dụng bởi hàng trăm nghìn lập trình viên trong mọi lĩnh vực ứng dụng.

C++ đang được sử dụng nhiều để viết trình điều khiển thiết bị và phần mềm khác dựa trên thao tác trực tiếp với phần cứng dưới các ràng buộc thời gian thực.

C++ được sử dụng rộng rãi cho giảng dạy và nghiên cứu vì nó đủ sạch để giảng dạy thành công các khái niệm cơ bản.

Bất kỳ ai đã sử dụng Apple Macintosh hoặc PC chạy Windows đều đã gián tiếp sử dụng C++ vì giao diện người dùng chính của các hệ thống này được viết bằng C++.

# THIẾT LẬP MÔI TRƯỜNG

## **Cài đặt môi trường cục bộ**

Nếu bạn vẫn sẵn sàng thiết lập môi trường cho C++, bạn cần có hai phần mềm sau trên máy tính của mình.

### **Text Editor (chương trình soạn thảo mã)**

Điều này sẽ được sử dụng để gõ chương trình của bạn. Ví dụ về một số trình chỉnh sửa bao gồm Windows Notepad, lệnh Chỉnh sửa hệ điều hành, Tóm tắt, Epsilon, EMACS và vim hoặc vi.

Tên và phiên bản của trình soạn thảo văn bản có thể khác nhau trên các hệ điều hành khác nhau. Ví dụ: Notepad sẽ được sử dụng trên Windows và vim hoặc vi có thể được sử dụng trên windows cũng như Linux hoặc UNIX.

Các tệp bạn tạo bằng trình chỉnh sửa của mình được gọi là tệp nguồn và đối với C++, chúng thường được đặt tên bằng phần mở rộng .cpp, .cp hoặc .c.

Cần có một trình soạn thảo văn bản để bắt đầu lập trình C++ của bạn.

### **C++ Compiler (chương trình biên dịch, liên kết)**

Đây là trình biên dịch C++ thực tế, sẽ được sử dụng để biên dịch mã nguồn của bạn thành chương trình thực thi cuối cùng.

Hầu hết các trình biên dịch C++ không quan tâm bạn cung cấp phần mở rộng nào cho mã nguồn của mình, nhưng nếu bạn không chỉ định cách khác, nhiều trình biên dịch sẽ sử dụng .cpp theo mặc định.

Trình biên dịch miễn phí và được sử dụng thường xuyên nhất là trình biên dịch GNU C/C++, nếu không, bạn có thể có trình biên dịch từ HP hoặc Solaris, msvc nếu bạn có Hệ điều hành tương ứng.

### **Cài đặt trình biên dịch GNU C/C++**

### **Cài đặt trên hệ điều hành họ UNIX/Linux**

Nếu bạn đang sử dụng Linux hoặc UNIX thì hãy kiểm tra xem GCC đã được cài đặt trên hệ thống của bạn hay chưa bằng cách nhập lệnh sau từ dòng lệnh –

$ g++ -v

Nếu bạn đã cài đặt GCC, thì nó sẽ in một thông báo như sau −

Using built-in specs.

Target: i386-redhat-linux

Configured with: ../configure --prefix=/usr .......

Thread model: posix

gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)

Nếu GCC chưa được cài đặt, thì bạn sẽ phải tự cài đặt nó bằng hướng dẫn chi tiết có sẵn tại https://gcc.gnu.org/install/

### **Cài đặt trên MAC OSMac OS**

Nếu bạn sử dụng Mac OS X, cách dễ nhất để lấy GCC là tải xuống môi trường phát triển Xcode từ trang web của Apple và làm theo hướng dẫn cài đặt đơn giản.

Xcode hiện có sẵn tại developer.apple.com/technologies/tools/.

### **Cài đặt trên Windows**

Để cài đặt GCC trên Windows, bạn cần cài đặt MinGW. Để cài đặt MinGW, hãy truy cập trang chủ MinGW, www.mingw.org và theo liên kết đến trang tải xuống MinGW. Tải xuống phiên bản mới nhất của chương trình cài đặt MinGW có tên là MinGW-<version>.exe.

Trong khi cài đặt MinGW, ở mức tối thiểu, bạn phải cài đặt gcc-core, gcc-g++, binutils và thời gian chạy MinGW, nhưng bạn có thể muốn cài đặt thêm.

Thêm thư mục con bin của bản cài đặt MinGW vào biến môi trường PATH để bạn có thể chỉ định các công cụ này trên dòng lệnh bằng tên đơn giản của chúng.

Khi quá trình cài đặt hoàn tất, bạn sẽ có thể chạy gcc, g++, ar, ranlib, dlltool và một số công cụ GNU khác từ dòng lệnh của Windows.

Tuy nhiên, trên windows có thể dùng một trình biên dịch khác tên là MSVC đi theo Vistual Studio IDE. Trình biên dịch này cũng có chất lượng tốt và thường được ưu tiên sử dụng trên Windows. Do vậy chúng ta cũng nên để ý đến trình biên dịch này.

# CÚ PHÁP CƠ BẢN

Khi chúng ta xem xét một chương trình C++, nó có thể được định nghĩa là một tập hợp các đối tượng giao tiếp thông qua việc gọi các phương thức của nhau. Bây giờ chúng ta hãy xem xét ngắn gọn ý nghĩa của một lớp, đối tượng, các phương thức và các biến tức thời.

* Đối tượng − Đối tượng có trạng thái và hành vi. Ví dụ: Một con chó có các trạng thái - màu sắc, tên, giống cũng như các hành vi - vẫy, sủa, ăn. Một đối tượng là một thể hiện của một lớp.
* Lớp − Một lớp có thể được định nghĩa là một mẫu/bản thiết kế mô tả các hành vi/trạng thái mà đối tượng thuộc loại của nó hỗ trợ.
* Phương thức − Một phương thức về cơ bản là một hành vi. Một lớp có thể chứa nhiều phương thức. Đó là trong các phương thức mà logic được viết, dữ liệu được thao tác và tất cả các hành động được thực thi.
* Biến thể hiện − Mỗi đối tượng có một bộ biến thể hiện duy nhất. Trạng thái của một đối tượng được tạo bởi các giá trị được gán cho các biến thể hiện này.

## **Cấu trúc một chương trình C++**

Chúng ta hãy xem một đoạn mã đơn giản sẽ in ra dòng chữ Hello World.

#include <iostream>

using namespace std;

// main() is where program execution begins.

int main() {

cout << "Hello World"; // prints Hello World

return 0;

}

Chúng ta hãy xem xét các phần khác nhau của chương trình trên:

* Ngôn ngữ C++ định nghĩa một số tiêu đề chứa thông tin cần thiết hoặc hữu ích cho chương trình của bạn. Đối với chương trình này, tiêu đề <iostream> là cần thiết.
* Dòng using namespace std; báo cho trình biên dịch sử dụng không gian tên std. Không gian tên là một bổ sung tương đối gần đây cho C++.
* Dòng tiếp theo '// main() là nơi bắt đầu thực hiện chương trình.' là chú thích một dòng có sẵn trong C++. Nhận xét một dòng bắt đầu bằng // và dừng ở cuối dòng.
* Dòng int main() là hàm chính bắt đầu thực thi chương trình.
* Dòng tiếp theo cout<<"Hello World"; làm cho thông báo "Xin chào thế giới" được hiển thị trên màn hình.
* Dòng tiếp theo trả về 0; kết thúc hàm main( ) và khiến nó trả về giá trị 0 cho quá trình gọi.

## **Biên dịch và thực thi chương trình C++**

* Lưu file với tên: hello.cpp
* Mở dấu nhắc lệnh và đi tới thư mục mà bạn đã lưu tệp.
* Nhập 'g++ hello.cpp' và nhấn enter để biên dịch mã của bạn. Nếu không có lỗi trong mã của bạn, dấu nhắc lệnh sẽ đưa bạn đến dòng tiếp theo và sẽ tạo tệp thực thi a.out.
* Bây giờ, gõ 'a.out' để chạy chương trình của bạn.
* Bạn sẽ có thể nhìn thấy ' Hello World ' được in trên cửa sổ terminal.

$ g++ hello.cpp

$ ./a.out

Hello World

Đảm bảo rằng g++ nằm trong đường dẫn của bạn và bạn đang chạy nó trong thư mục chứa tệp hello.cpp.

Bạn có thể biên dịch các chương trình C/C++ bằng makefile.

Hiện nay, có nhiều IDE hỗ trợ C++. Trong các IDE này chúng ta chỉ cần chọn menu hoặc nút Build/ Build and Run để tự động hóa các công đoạn trên.

## **Dấu chấm phẩy (;) và khối lệnh trong C++**

Trong C++, dấu chấm phẩy (;) là một ký tự để kế thúc một câu lệnh. Nghĩa là mỗi một câu lệnh riêng lẻ sẽ phải được kết thúc bằng dấu “;”.

Ví dụ bên dưới có 3 câu lệnh

x = y;

y = y + 1;

add(x, y);

Một khối lệnh (block) là tập hợp của nhiều câu lệnh được bao quanh bởi dấu “{“ và “}”.

Ví dụ:

{

cout << "Hello World"; // prints Hello World

return 0;

}

Trong C++, nó không thể nhận dạng được kết thúc câu lệnh bằng dấu xuống dòng. Do vậy sau mỗi câu lệnh phải có dấu “;”, đồng thời vị trí câu lệnh ở đâu không quan trọng.

Ví dụ :

x = y;

y = y + 1;

add(x, y);

cũng tương đương với

x = y; y = y + 1; add(x, y);

## **Định danh trong C++**

Một định danh trong C++ là một cái tên để đặt cho một biến, hàm, lớp, module hoặc nhiều thứ khác mà người dùng khai báo. Một định danh được bắt đầu bằng một chữ cái trong khoảng A-Z hoặc a-z hoặc một dấu shift trừ (underscore) (\_). Tiếp sau đó có thể là nhiều chữ cái, dấu “\_” hoặc số (0-9).

C++ không cho phép sử dụng các ký tự đặc biệt như @,$,% để tạo nên định danh và có phân biệt các chữ hoa và chữ thường.

Đây là một số ví dụ về định danh được chấp nhận (hợp lệ):

mohd

zara

abc

move\_name

a\_123

myname50

\_temp

j

a23b9

retVal

## **Keyword trong C++**

Danh sách sau đây hiển thị các từ dành riêng trong C++. Những từ dành riêng này có thể không được sử dụng làm hằng số hoặc biến hoặc bất kỳ tên định danh nào khác.

|  |  |  |  |
| --- | --- | --- | --- |
| asm | else | new | this |
| auto | enum | operator | throw |
| bool | explicit | private | true |
| break | export | protected | try |
| case | extern | public | typedef |
| catch | false | register | typeid |
| char | float | reinterpret\_cast | typename |
| class | for | return | union |
| const | friend | short | unsigned |
| const\_cast | goto | signed | using |
| continue | if | sizeof | virtual |
| default | inline | static | void |
| delete | int | static\_cast | volatile |
| do | long | struct | wchar\_t |
| double | mutable | switch | while |
| dynamic\_cast | namespace | template |  |

## **Tam đồ**

Một vài ký tự có một biểu diễn thay thế, được gọi là chuỗi ba ký tự. Trigraph là một chuỗi ba ký tự đại diện cho một ký tự duy nhất và chuỗi luôn bắt đầu bằng hai dấu chấm hỏi.

Trigraphs được mở rộng ở bất cứ nơi nào chúng xuất hiện, bao gồm cả trong chuỗi ký tự và ký tự ký tự, trong nhận xét và trong chỉ thị tiền xử lý.

Sau đây là các trình tự trigraph được sử dụng thường xuyên nhất

|  |  |
| --- | --- |
| **Trigraph** | **Replacement** |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

Không phải tất cả trình biên dịch đều hỗ trợ tam đồ và họ khuyên không nên sử dụng bởi vì chúng có thể gây nên các nhầm lẫn.

## **Khoảng trắng trong C++**

Một dòng chỉ chứa khoảng trắng, có thể có chú thích, được gọi là dòng trống và trình biên dịch C++ hoàn toàn bỏ qua nó.

Khoảng trắng là thuật ngữ được sử dụng trong C++ để mô tả khoảng trống, tab, ký tự dòng mới và nhận xét. Khoảng trắng phân tách một phần của câu lệnh với phần khác và cho phép trình biên dịch xác định vị trí của một phần tử trong câu lệnh, chẳng hạn như int, kết thúc và phần tử tiếp theo bắt đầu.

Ví dụ 1:

int age;

Trong câu lệnh trên phải có ít nhất một ký tự khoảng trắng (thường là khoảng trắng) giữa int và age để trình biên dịch có thể phân biệt chúng.

Ví dụ 2:

fruit = apples + oranges; // Get the total fruit

Trong câu lệnh 2 ở trên, không cần ký tự khoảng trắng giữa trái cây và =, hoặc giữa = và táo, mặc dù bạn có thể tự do thêm một số ký tự này nếu muốn vì mục đích dễ đọc.

# CHÚ THÍCH

Chú thích chương trình là những câu giải thích mà bạn có thể đưa vào mã C++. Những nhận xét này giúp bất kỳ ai đọc mã nguồn. Tất cả các ngôn ngữ lập trình đều cho phép một số dạng nhận xét.

C++ hỗ trợ chú thích một dòng và nhiều dòng. Tất cả các ký tự có sẵn bên trong bất kỳ nhận xét nào đều bị trình biên dịch C++ bỏ qua.

Nhận xét C++ bắt đầu bằng /\* và kết thúc bằng \*/. Ví dụ :

/\* This is a comment \*/

/\* C++ comments can also

\* span multiple lines

\*/

Chú thích cũng có thể bắt đầu bằng //, kéo dài đến cuối dòng. Ví dụ :

#include <iostream>

using namespace std;

main() {

cout << "Hello World"; // prints Hello World

return 0;

}

Khi đoạn mã trên được biên dịch, nó sẽ bỏ qua // in Hello World và phần thực thi cuối cùng sẽ tạo ra kết quả sau -

Hello world

Trong một nhận xét /\* và \*/, các ký tự // không có ý nghĩa đặc biệt. Trong một chú thích //, /\* và \*/ không có ý nghĩa đặc biệt. Vì vậy, bạn có thể "lồng" một loại nhận xét trong loại khác. Ví dụ −

/\* Comment out printing of Hello World:

cout << "Hello World"; // prints Hello World

\*/

# KIỂU DỮ LIỆU

Khi viết chương trình bằng bất kỳ ngôn ngữ nào, bạn cần sử dụng nhiều biến khác nhau để lưu trữ các thông tin khác nhau. Các biến không là gì ngoài các vị trí bộ nhớ dành riêng để lưu trữ các giá trị. Điều này có nghĩa là khi bạn tạo một biến, bạn dành một khoảng trống trong bộ nhớ.

Bạn có thể muốn lưu trữ thông tin thuộc nhiều loại dữ liệu khác nhau như ký tự, ký tự rộng, số nguyên, dấu phẩy động, dấu phẩy động kép, boolean, v.v. Dựa trên loại dữ liệu của một biến, hệ điều hành sẽ phân bổ bộ nhớ và quyết định những gì có thể được lưu trữ trong bộ nhớ dành riêng.

## **Các kiểu dữ liệu tích hợp nguyên thủy**

(Primitive Built-in Types)

C++ cung cấp cho người lập trình nhiều loại dữ liệu có sẵn cũng như do người dùng định nghĩa. Bảng sau liệt kê bảy kiểu dữ liệu cơ bản của C++

|  |  |
| --- | --- |
| **Type** | **Keyword** |
| Boolean | bool |
| Character | char |
| Integer | int |
| Floating point | float |
| Double floating point | double |
| Valueless | void |
| Wide character | wchar\_t |

Một số loại cơ bản có thể được sửa đổi bằng cách sử dụng một hoặc nhiều công cụ sửa đổi loại này :

* signed
* unsigned
* short
* long

Bảng sau đây cho biết loại biến, cần bao nhiêu bộ nhớ để lưu trữ giá trị trong bộ nhớ và giá trị tối đa và tối thiểu có thể được lưu trữ trong loại biến đó là bao nhiêu.

|  |  |  |
| --- | --- | --- |
| **Type** | **Typical Bit Width** | **Typical Range** |
| char | 1byte | -127 to 127 or 0 to 255 |
| unsigned char | 1byte | 0 to 255 |
| signed char | 1byte | -127 to 127 |
| int | 4bytes | -2147483648 to 2147483647 |
| unsigned int | 4bytes | 0 to 4294967295 |
| signed int | 4bytes | -2147483648 to 2147483647 |
| short int | 2bytes | -32768 to 32767 |
| unsigned short int | 2bytes | 0 to 65,535 |
| signed short int | 2bytes | -32768 to 32767 |
| long int | 8bytes | -9223372036854775808 to 9223372036854775807 |
| signed long int | 8bytes | same as long int |
| unsigned long int | 8bytes | 0 to 18446744073709551615 |
| long long int | 8bytes | -(2^63) to (2^63)-1 |
| unsigned long long int | 8bytes | 0 to 18,446,744,073,709,551,615 |
| float | 4bytes |  |
| double | 8bytes |  |
| long double | 12bytes |  |
| wchar\_t | 2 or 4 bytes | 1 wide character |

Kích thước của các biến có thể khác với kích thước được hiển thị trong bảng trên, tùy thuộc vào trình biên dịch và máy tính bạn đang sử dụng.

Sau đây là ví dụ sẽ tạo ra kích thước chính xác của các loại dữ liệu khác nhau trên máy tính của bạn.

#include <iostream>

using namespace std;

int main() {

cout << "Size of char : " << sizeof(char) << endl;

cout << "Size of int : " << sizeof(int) << endl;

cout << "Size of short int : " << sizeof(short int) << endl;

cout << "Size of long int : " << sizeof(long int) << endl;

cout << "Size of float : " << sizeof(float) << endl;

cout << "Size of double : " << sizeof(double) << endl;

cout << "Size of wchar\_t : " << sizeof(wchar\_t) << endl;

return 0;

}

Ví dụ này sử dụng endl, chèn một ký tự dòng mới sau mỗi dòng và toán tử << đang được sử dụng để chuyển nhiều giá trị ra màn hình. Chúng tôi cũng đang sử dụng toán tử sizeof() để lấy kích thước của các loại dữ liệu khác nhau.

Khi đoạn mã trên được biên dịch và thực thi, nó sẽ tạo ra kết quả sau, kết quả này có thể khác nhau giữa các máy

Size of char : 1

Size of int : 4

Size of short int : 2

Size of long int : 4

Size of float : 4

Size of double : 8

Size of wchar\_t : 4

Sau đây là một ví dụ khác:

#include <iostream>

#include <limits>

using namespace std;

int main() {

std::cout << "Int Min " << std::numeric\_limits<int>::min() << endl;

std::cout << "Int Max " << std::numeric\_limits<int>::max() << endl;

std::cout << "Unsigned Int Min " << std::numeric\_limits<unsigned int>::min() << endl;

std::cout << "Unsigned Int Max " << std::numeric\_limits<unsigned int>::max() << endl;

std::cout << "Long Int Min " << std::numeric\_limits<long int>::min() << endl;

std::cout << "Long Int Max " << std::numeric\_limits<long int>::max() << endl;

std::cout << "Unsigned Long Int Min " << std::numeric\_limits<unsigned long int>::min() <<endl;

std::cout << "Unsigned Long Int Max " << std::numeric\_limits<unsigned long int>::max() << endl;

}

## **Khai báo typedef**

Bạn có thể tạo một tên mới cho một loại hiện có bằng cách sử dụng typedef. Sau đây là cú pháp đơn giản để định nghĩa một kiểu mới bằng cách sử dụng typedef :

typedef type newname;

Ví dụ, phần sau cho trình biên dịch biết rằng feet là một tên khác của int :

typedef int feet;

Bây giờ, khai báo sau đây là hoàn toàn hợp lệ và tạo ra một biến số nguyên gọi là distance:

feet distance;

## **Kiểu liệt kê**

Kiểu liệt kê khai báo một tên kiểu tùy chọn và một tập hợp không hoặc nhiều số nhận dạng có thể được sử dụng làm giá trị của kiểu. Mỗi điều tra viên là một hằng số có kiểu liệt kê.

Tạo một bảng liệt kê yêu cầu sử dụng từ khóa enum. Dạng tổng quát của một kiểu liệt kê là :

enum enum-name { list of names } var-list;

Ở đây, tên enum là tên kiểu của phép liệt kê. Danh sách tên được phân tách bằng dấu phẩy.

Ví dụ: đoạn mã sau định nghĩa một bảng liệt kê các màu được gọi là colors và biến c thuộc loại color. Cuối cùng, c được gán giá trị "blue".

enum color {

red,

green,

blue

} c;

c = blue;

Theo mặc định, giá trị của tên đầu tiên là 0, tên thứ hai có giá trị 1 và tên thứ ba có giá trị 2, v.v. Nhưng bạn có thể đặt tên, giá trị cụ thể bằng cách thêm bộ khởi tạo. Ví dụ, trong bảng liệt kê sau, màu xanh lá cây sẽ có giá trị 5.

enum color {

red,

green = 5,

blue

};

Ở đây, màu xanh lam sẽ có giá trị là 6 vì mỗi tên sẽ lớn hơn tên đứng trước nó một giá trị.

# KIỂU BIẾN

Một biến cung cấp cho chúng tôi bộ lưu trữ được đặt tên mà các chương trình của chúng tôi có thể thao tác. Mỗi biến trong C++ có một kiểu xác định, xác định kích thước và cách bố trí bộ nhớ của biến; phạm vi giá trị có thể được lưu trữ trong bộ nhớ đó; và tập hợp các hoạt động có thể được áp dụng cho biến.

Tên của một biến có thể bao gồm các chữ cái, chữ số và ký tự gạch dưới. Nó phải bắt đầu bằng một chữ cái hoặc dấu gạch dưới. Chữ hoa và chữ thường khác nhau vì C++ phân biệt chữ hoa chữ thường.

Có các loại biến cơ bản sau trong C++ như đã giải thích trong chương trước :

|  |  |
| --- | --- |
| **Sr.No** | **Type & Description** |
| 1 | **bool**  Lưu trữ giá trị true hoặc false |
| 2 | **char**  Một byte đơn, là một kiểu số nguyên |
| 3 | **int**  Một số nguyên 16 hoặc 32 bit. |
| 4 | **float**  Một kiểu số thực 32 bit. |
| 5 | **double**  Một kiểu số thực 64 bit, chính xác hơn float. |
| 6 | **void**  Không xác định loại. |
| 7 | **wchar\_t**  Ký tự rộng. |

C++ cũng cho phép định nghĩa nhiều loại biến khác mà chúng ta sẽ đề cập trong các chương tiếp theo như Liệt kê, Con trỏ, Mảng, Tham chiếu, Cấu trúc dữ liệu và Lớp.

Phần sau sẽ trình bày cách định nghĩa, khai báo và sử dụng các loại biến.

## **Khai báo biến trong C++**

Một định nghĩa biến cho trình biên dịch biết vị trí và dung lượng lưu trữ cần tạo cho biến. Một định nghĩa biến chỉ định một kiểu dữ liệu và chứa danh sách một hoặc nhiều biến của kiểu đó như sau:

type variable\_list;

Ở đây, loại phải là loại dữ liệu C++ hợp lệ bao gồm char, w\_char, int, float, double, bool hoặc bất kỳ đối tượng nào do người dùng định nghĩa, v.v. và danh\_sách\_biến có thể bao gồm một hoặc nhiều tên định danh được phân tách bằng dấu phẩy. Một số khai báo hợp lệ được hiển thị ở đây:

int i, j, k;

char c, ch;

float f, salary;

double d;

Dòng int i, j, k; vừa khai báo vừa định nghĩa các biến i, j và k; hướng dẫn trình biên dịch tạo các biến có tên i, j và k kiểu int.

Các biến có thể được khởi tạo (gán giá trị ban đầu) trong khai báo của chúng. Trình khởi tạo bao gồm một dấu bằng theo sau là biểu thức hằng số như sau :

type variable\_name = value;

Một số ví dụ:

extern int d = 3, f = 5; // declaration of d and f.

int d = 3, f = 5; // definition and initializing d and f.

byte z = 22; // definition and initializes z.

char x = 'x'; // the variable x has the value 'x'.

Đối với định nghĩa không có bộ khởi tạo: các biến có thời lượng lưu trữ tĩnh được khởi tạo hoàn toàn bằng NULL (tất cả các byte có giá trị 0); giá trị ban đầu của tất cả các biến khác là không xác định.

## **Xác minh biến trong C++**

Một khai báo biến cung cấp sự đảm bảo cho trình biên dịch rằng có một biến tồn tại với kiểu và tên đã cho để trình biên dịch tiếp tục biên dịch mà không cần chi tiết đầy đủ về biến đó. Một khai báo biến chỉ có ý nghĩa tại thời điểm biên dịch, trình biên dịch cần định nghĩa biến thực tại thời điểm liên kết chương trình.

Khai báo biến hữu ích khi bạn đang sử dụng nhiều tệp và bạn xác định biến của mình trong một trong các tệp sẽ có sẵn tại thời điểm liên kết chương trình. Bạn sẽ sử dụng từ khóa bên ngoài để khai báo một biến ở bất kỳ đâu. Mặc dù bạn có thể khai báo một biến nhiều lần trong chương trình C++ nhưng nó chỉ có thể được định nghĩa một lần trong một tệp, một hàm hoặc một khối mã.

Ví dụ :

Hãy thử ví dụ sau khi một biến đã được khai báo ở trên cùng, nhưng nó đã được định nghĩa bên trong hàm chính :

#include <iostream>

using namespace std;

// Variable declaration:

extern int a, b;

extern int c;

extern float f;

int main () {

// Variable definition:

int a, b;

int c;

float f;

// actual initialization

a = 10;

b = 20;

c = a + b;

cout << c << endl ;

f = 70.0/3.0;

cout << f << endl ;

return 0;

}

Khi đoạn mã trên được biên dịch và thực thi, nó sẽ tạo ra kết quả sau :

30

23.3333

Khái niệm tương tự áp dụng cho khai báo hàm trong đó bạn cung cấp tên hàm tại thời điểm khai báo và định nghĩa thực tế của nó có thể được cung cấp ở bất kỳ nơi nào khác. Ví dụ :

// function declaration

int func();

int main() {

// function call

int i = func();

}

// function definition

int func() {

return 0;

}

## **Lvalues và Rvalues**

Có hai loại biểu thức trong C++:

* + lvalue − Biểu thức tham chiếu đến một vị trí bộ nhớ được gọi là biểu thức "lvalue". Một giá trị có thể xuất hiện ở bên trái hoặc bên phải của một phép gán.
  + rvalue − Thuật ngữ rvalue đề cập đến một giá trị dữ liệu được lưu trữ tại một số địa chỉ trong bộ nhớ. Một giá trị là một biểu thức không thể có một giá trị được gán cho nó, điều đó có nghĩa là một giá trị có thể xuất hiện ở phía bên phải nhưng không xuất hiện ở phía bên trái của một phép gán.

Các biến là các giá trị và do đó có thể xuất hiện ở phía bên trái của một phép gán. Chữ số là giá trị và do đó có thể không được chỉ định và không thể xuất hiện ở phía bên trái. Sau đây là một tuyên bố hợp lệ :

int g = 20;

Nhưng sau đây không phải là một câu lệnh hợp lệ và sẽ tạo ra lỗi thời gian biên dịch :

10 = 20;

# PHẠM VI CỦA BIẾN

Phạm vi là một vùng của chương trình và nói chung có ba vị trí, nơi các biến có thể được khai báo :

* Bên trong một hàm hoặc một khối được gọi là biến cục bộ,
* Trong định nghĩa các tham số của hàm được gọi là tham số hình thức.
* Ngoài tất cả các hàm được gọi là biến toàn cục.

Chúng ta sẽ tìm hiểu hàm là gì và tham số của nó trong các chương tiếp theo. Ở đây hãy để chúng tôi giải thích các biến cục bộ và toàn cục là gì.

## **Biến cục bộ**

(Local Variables)

Các biến được khai báo bên trong hàm hoặc khối là biến cục bộ. Chúng chỉ có thể được sử dụng bởi các câu lệnh bên trong hàm hoặc khối mã đó. Các biến cục bộ không được biết đến với các hàm bên ngoài của chúng. Sau đây là ví dụ sử dụng biến cục bộ:

[Live Demo](http://tpcg.io/QIjnPh)

#include <iostream>

using namespace std;

int main () {

// Local variable declaration:

int a, b;

int c;

// actual initialization

a = 10;

b = 20;

c = a + b;

cout << c;

return 0;

}

## **Biến toàn cục**

(Global Variables)

Các biến toàn cục được định nghĩa bên ngoài tất cả các hàm, thường ở đầu chương trình. Các biến toàn cầu sẽ giữ giá trị của chúng trong suốt thời gian tồn tại của chương trình của bạn.

Một biến toàn cục có thể được truy cập bởi bất kỳ hàm nào. Nghĩa là, một biến toàn cục có sẵn để sử dụng trong toàn bộ chương trình của bạn sau khi khai báo. Sau đây là ví dụ sử dụng biến toàn cục và biến cục bộ :

#include <iostream>

using namespace std;

// Global variable declaration:

int g;

int main () {

// Local variable declaration:

int a, b;

// actual initialization

a = 10;

b = 20;

g = a + b;

cout << g;

return 0;

}

Một chương trình có thể có cùng tên cho các biến cục bộ và toàn cầu nhưng giá trị của biến cục bộ bên trong một hàm sẽ được ưu tiên. Ví dụ :

#include <iostream>

using namespace std;

// Global variable declaration:

int g = 20;

int main () {

// Local variable declaration:

int g = 10;

cout << g;

return 0;

}

Khi đoạn mã trên được biên dịch và thực thi, nó sẽ tạo ra kết quả sau :

10

Khi đoạn mã trên được biên dịch và thực thi, nó sẽ tạo ra kết quả sau

## **Khởi tạo các biến cục bộ và toàn cục**

Khi một biến cục bộ được xác định, nó không được khởi tạo bởi hệ thống, bạn phải tự khởi tạo nó. Các biến toàn cục được hệ thống khởi tạo tự động khi bạn định nghĩa chúng như sau:

|  |  |
| --- | --- |
| **Data Type** | **Initializer** |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

Đó là một thực hành lập trình tốt để khởi tạo các biến đúng cách, nếu không, đôi khi chương trình sẽ tạo ra kết quả không mong muốn.

# HẰNG SỐ / THỂ HIỆN

Hằng đề cập đến các giá trị cố định mà chương trình không thể thay đổi và chúng được gọi là hằng.

Các hằng số có thể thuộc bất kỳ loại dữ liệu cơ bản nào và có thể được chia thành Số nguyên, Số dấu phẩy động, Ký tự, Chuỗi và Giá trị Boolean.

Một lần nữa, các hằng số được xử lý giống như các biến thông thường ngoại trừ việc giá trị của chúng không thể được sửa đổi sau định nghĩa của chúng.

## **Thể hiện số nguyên**

Một số nguyên có thể là hằng số thập phân, bát phân hoặc thập lục phân. Tiền tố chỉ định cơ số hoặc cơ số: 0x hoặc 0X cho hệ thập lục phân, 0 cho hệ bát phân và không có gì cho hệ thập phân.

Một chữ số nguyên cũng có thể có một hậu tố là sự kết hợp của U và L, tương ứng cho không dấu và dài. Hậu tố có thể là chữ hoa hoặc chữ thường và có thể theo thứ tự bất kỳ.

Dưới đây là một số ví dụ về số nguyên :

212 // Legal

215u // Legal

0xFeeL // Legal

078 // Illegal: 8 is not an octal digit

032UU // Illegal: cannot repeat a suffix

Following are other examples of various types of Integer literals −

85 // decimal

0213 // octal

0x4b // hexadecimal

30 // int

30u // unsigned int

30l // long

30ul // unsigned long

## **Thể hiện số thực float**

Một chữ số dấu phẩy động có một phần nguyên, một dấu thập phân, một phần phân số và một phần số mũ. Bạn có thể biểu diễn các chữ số dấu phẩy động ở dạng thập phân hoặc dạng hàm mũ.

Khi biểu diễn bằng dạng thập phân, bạn phải bao gồm dấu thập phân, số mũ hoặc cả hai và khi biểu diễn bằng dạng hàm mũ, bạn phải bao gồm phần nguyên, phần phân số hoặc cả hai. Số mũ có dấu được giới thiệu bởi e hoặc E.

Dưới đây là một số ví dụ về chữ dấu phẩy động:

3.14159 // Legal

314159E-5L // Legal

510E // Illegal: incomplete exponent

210f // Illegal: no decimal or exponent

.e55 // Illegal: missing integer or fraction

## **Thể hiện boolean**

Có hai chữ Boolean và chúng là một phần của từ khóa C++ tiêu chuẩn :

* Một giá trị của true đại diện cho đúng.
* Một giá trị false đại diện cho sai.

## **Thể hiện ký tự**

Chữ ký tự được đặt trong dấu nháy đơn. Nếu chữ bắt đầu bằng L (chỉ viết hoa), thì đó là chữ có ký tự rộng (ví dụ: L'x') và phải được lưu trữ trong loại biến wchar\_t . Mặt khác, nó là một ký tự hẹp (ví dụ: 'x') và có thể được lưu trữ trong một biến đơn giản kiểu char.

Một ký tự chữ có thể là một ký tự đơn giản (ví dụ: 'x'), một chuỗi thoát (ví dụ: '\t') hoặc một ký tự chung (ví dụ: '\u02C0').

Có một số ký tự trong C++ khi đứng trước dấu gạch chéo ngược sẽ có ý nghĩa đặc biệt và được dùng để biểu diễn như xuống dòng (\n) hoặc tab (\t). Ở đây, bạn có một danh sách một số mã trình tự thoát như vậy :

|  |  |
| --- | --- |
| **Escape sequence** | **Meaning** |
| \\ | \ character |
| \' | ' character |
| \" | " character |
| \? | ? character |
| \a | Alert or bell |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \ooo | Octal number of one to three digits |
| \xhh . . . | Hexadecimal number of one or more digits |

Sau đây là ví dụ để hiển thị một vài ký tự chuỗi thoát :

#include <iostream>

using namespace std;

int main() {

cout << "Hello\tWorld\n\n";

return 0;

}

When the above code is compiled and executed, it produces the following result :

Hello World

## **Thể hiện chuỗi**

Chuỗi ký tự được đặt trong dấu ngoặc kép. Một chuỗi chứa các ký tự tương tự như ký tự chữ: ký tự đơn giản, chuỗi thoát và ký tự phổ quát.

Bạn có thể ngắt một dòng dài thành nhiều dòng bằng cách sử dụng chuỗi ký tự và phân tách chúng bằng khoảng trắng.

Dưới đây là một số ví dụ về chuỗi ký tự. Tất cả ba hình thức là chuỗi giống hệt nhau.

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

## **Khai báo hằng số**

Có 2 cách đơn giản trong C++ để khai báo một hằng số :

* Sử dụng tiền xử lý **#define**.
* Sử dụng từ khóa **const**.

## **Tiền xử lý #define**

Sử dụng cú pháp #define preprocessor để khai báo một hằng số

#define identifier value

Following example explains it in detail −

#include <iostream>

using namespace std;

#define LENGTH 10

#define WIDTH 5

#define NEWLINE '\n'

int main() {

int area;

area = LENGTH \* WIDTH;

cout << area;

cout << NEWLINE;

return 0;

}

Khi cả đoạn mã trên được biên dịch và thực thi, nó cho ra kết quả :

50

## **Từ khóa const**

Bạn có thể sử dụng từ khóa const để khai báo hằng số với một kiểu dữ liệu cụ thể:

const type variable = value;

Ví dụ:

#include <iostream>

using namespace std;

int main() {

const int LENGTH = 10;

const int WIDTH = 5;

const char NEWLINE = '\n';

int area;

area = LENGTH \* WIDTH;

cout << area;

cout << NEWLINE;

return 0;

}

Khi đoạn code thực thi sẽ cho kết quả :

50

Chú ý : các hằng số nên được đặt tên là chữ hoa toàn bộ.

# MODIFIER TYPES

C++ cho phép các kiểu dữ liệu char, int và double có các bộ sửa đổi trước chúng. Một công cụ sửa đổi được sử dụng để thay đổi ý nghĩa của loại cơ sở sao cho nó phù hợp chính xác hơn với nhu cầu của các tình huống khác nhau.

Các công cụ sửa đổi kiểu dữ liệu được liệt kê ở đây :

* signed
* unsigned
* long
* short

Các công cụ sửa đổi có dấu, không dấu, dài và ngắn có thể được áp dụng cho các kiểu cơ sở số nguyên. Ngoài ra, có thể áp dụng có dấu và không dấu cho char, và long có thể được áp dụng cho double.

Các công cụ sửa đổi đã ký và chưa ký cũng có thể được sử dụng làm tiền tố cho các công cụ sửa đổi dài hoặc ngắn. Ví dụ: int dài không dấu.

C++ cho phép ký hiệu tốc ký để khai báo các số nguyên không dấu, ngắn hoặc dài. Bạn chỉ có thể sử dụng từ không dấu, ngắn hoặc dài mà không cần int. Nó tự động ngụ ý int. Ví dụ, hai câu lệnh sau đều khai báo biến số nguyên không dấu.

unsigned x;

unsigned int y;

Để hiểu sự khác biệt giữa cách diễn giải các biến số nguyên có dấu và không dấu trong C++, bạn nên chạy chương trình ngắn sau :

#include <iostream>

using namespace std;

/\* This program shows the difference between

\* signed and unsigned integers.

\*/

int main() {

short int i; // a signed short integer

short unsigned int j; // an unsigned short integer

j = 50000;

i = j;

cout << i << " " << j;

return 0;

}

Kết quả khi thực thi :

-15536 50000

Kết quả trên là do i là short int.

## **Type Qualifiers in C++**

The type qualifiers provide additional information about the variables they precede.

|  |  |
| --- | --- |
| **Sr.No** | **Qualifier & Meaning** |
| 1 | **const**  Objects of type **const** cannot be changed by your program during execution. |
| 2 | **volatile**  The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program. |
| 3 | **restrict**  A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict. |

# STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

* auto
* register
* static
* extern
* mutable

## **The auto Storage Class**

The **auto** storage class is the default storage class for all local variables.

{

int mount;

auto int month;

}

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

## **The register Storage Class**

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

{

register int miles;

}

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

## **The static Storage Class**

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

[Live Demo](http://tpcg.io/20HjiV)

#include <iostream>

// Function declaration

void func(void);

static int count = 10; /\* Global variable \*/

main() {

while(count--) {

func();

}

return 0;

}

// Function definition

void func( void ) {

static int i = 5; // local static variable

i++;

std::cout << "i is " << i ;

std::cout << " and count is " << count << std::endl;

}

When the above code is compiled and executed, it produces the following result −

i is 6 and count is 9

i is 7 and count is 8

i is 8 and count is 7

i is 9 and count is 6

i is 10 and count is 5

i is 11 and count is 4

i is 12 and count is 3

i is 13 and count is 2

i is 14 and count is 1

i is 15 and count is 0

## **The extern Storage Class**

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern*is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

### **First File: main.cpp**

#include <iostream>

int count ;

extern void write\_extern();

main() {

count = 5;

write\_extern();

}

### **Second File: support.cpp**

#include <iostream>

extern int count;

void write\_extern(void) {

std::cout << "Count is " << count << std::endl;

}

Here, *extern* keyword is being used to declare count in another file. Now compile these two files as follows −

$g++ main.cpp support.cpp -o write

This will produce **write** executable program, try to execute **write** and check the result as follows −

$./write

5

## **The mutable Storage Class**

The **mutable** specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function.

# TOÁN TỬ

Toán tử là một biểu tượng cho biết trình biên dịch thực hiện các thao tác logic hoặc toán học cụ thể. C++ có nhiều toán tử tích hợp sẵn và cung cấp các loại toán tử sau –

* Toán tử số học
* Toán tử quan hệ
* Toán tử logic
* Toán tử Bitwise
* Toán tử gán
* Toán tử khác

Chương này sẽ xem xét lần lượt các toán tử số học, quan hệ, logic, bitwise, phép gán và các toán tử khác.

## **Toán tử số học**

Có các toán tử số học sau đây được hỗ trợ bởi ngôn ngữ C++.

Giả sử biến A giữ 10 và biến B giữ 20, sau đó :

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Cộng 2 số | A + B = 30 |
| - | Trừ 2 số | A - B = -10 |
| \* | Nhân 2 số | A \* B = 200 |
| / | Chia 2 số lấy phần nguyên | B / A = 2 |
| % | Chia lấy dư | B % A = 0 |
| ++ | Tăng giá trị lên 1 | A++ = 11 |
| -- | Giảm giá trị xuống 1 | A-- = 9 |

## **Toán tử quan hệ**

Có các toán tử quan hệ sau được ngôn ngữ C++ hỗ trợ

Giả sử biến A giữ 10 và biến B giữ 20, sau đó:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Kiểm tra xem giá trị của hai toán hạng có bằng nhau hay không, nếu có thì điều kiện trở thành true. | (A == B) is not true. |
| != | Kiểm tra xem giá trị của hai toán hạng có bằng nhau hay không, nếu giá trị không bằng nhau thì điều kiện trở thành true. | (A != B) is true. |
| > | Kiểm tra xem giá trị của toán hạng bên trái có lớn hơn giá trị của toán hạng bên phải không, nếu có thì điều kiện trở thành true. | (A > B) is not true. |
| < | Kiểm tra xem giá trị của toán hạng bên trái có nhỏ hơn giá trị của toán hạng bên phải không, nếu có thì điều kiện trở thành true. | (A < B) is true. |
| >= | Kiểm tra xem giá trị của toán hạng bên trái có lớn hơn hoặc bằng giá trị của toán hạng bên phải hay không, nếu có thì điều kiện trở thành true. | (A >= B) is not true. |
| <= | Kiểm tra xem giá trị của toán hạng bên trái có nhỏ hơn hoặc bằng giá trị của toán hạng bên phải hay không, nếu có thì điều kiện trở thành true. | (A <= B) is true. |

## **Toán tử logic**

Có các toán tử logic sau đây được hỗ trợ bởi ngôn ngữ C++.

Giả sử biến A giữ giá trị 1 và biến B giữ giá trị 0, thì :

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Được gọi là toán tử logic AND. Nếu cả hai toán hạng khác không, thì điều kiện trở thành true. | (A && B) = false. |
| || | Được gọi là toán tử logic OR. Nếu bất kỳ toán hạng nào trong hai toán hạng khác không, thì điều kiện trở thành true. | (A || B) = true. |
| ! | Được gọi là toán tử logic NOT. Sử dụng để đảo ngược trạng thái logic của toán hạng của nó. Nếu một điều kiện là đúng, thì toán tử Logical NOT sẽ sai. | !(A && B) = true. |

## **Toán tử bit**

Toán tử bitwise hoạt động trên các bit và thực hiện thao tác từng bit. Các bảng chân trị cho &, |, và ^ như sau :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

Các toán tử Bitwise được hỗ trợ bởi ngôn ngữ C++ được liệt kê trong bảng sau. Giả sử biến A giữ 60 và biến B giữ 13, sau đó :

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Toán tử AND nhị phân sao chép một bit vào kết quả nếu nó tồn tại trong cả hai toán hạng. | (A & B) will give 12 which is 0000 1100 |
| | | Toán tử OR nhị phân sao chép một bit nếu nó tồn tại trong một trong hai toán hạng. | (A | B) will give 61 which is 0011 1101 |
| ^ | Toán tử XOR nhị phân sao chép bit nếu nó được đặt trong một toán hạng chứ không phải cả hai. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Toán tử bổ sung nhị phân là đơn nguyên và có tác dụng 'lật' các bit. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Toán tử dịch trái nhị phân. Giá trị toán hạng bên trái được di chuyển sang trái theo số bit được chỉ định bởi toán hạng bên phải. | A << 2 will give 240 which is 1111 0000 |
| >> | Toán tử dịch phải nhị phân. Giá trị toán hạng bên trái được di chuyển sang phải theo số bit được chỉ định bởi toán hạng bên phải. | A >> 2 will give 15 which is 0000 1111 |

## **Toán tử gán**

Có các toán tử gán sau được hỗ trợ bởi ngôn ngữ C++:

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand. | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand. | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | Bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

## **Toán tử khác**

The following table lists some other operators that C++ supports.

|  |  |
| --- | --- |
| **Sr.No** | **Operator & Description** |
| 1 | **sizeof**  [sizeof operator](https://www.tutorialspoint.com/cplusplus/cpp_sizeof_operator.htm) returns the size of a variable. For example, sizeof(a), where ‘a’ is integer, and will return 4. |
| 2 | **Condition ? X : Y**  [Conditional operator (?)](https://www.tutorialspoint.com/cplusplus/cpp_conditional_operator.htm). If Condition is true then it returns value of X otherwise returns value of Y. |
| 3 | **,**  [Comma operator](https://www.tutorialspoint.com/cplusplus/cpp_comma_operator.htm) causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list. |
| 4 | **. (dot) and -> (arrow)**  [Member operators](https://www.tutorialspoint.com/cplusplus/cpp_member_operators.htm) are used to reference individual members of classes, structures, and unions. |
| 5 | **Cast**  [Casting operators](https://www.tutorialspoint.com/cplusplus/cpp_casting_operators.htm) convert one data type to another. For example, int(2.2000) would return 2. |
| 6 | **&**  [Pointer operator &](https://www.tutorialspoint.com/cplusplus/cpp_pointer_operators.htm) returns the address of a variable. For example &a; will give actual address of the variable. |
| 7 | **\***  [Pointer operator \*](https://www.tutorialspoint.com/cplusplus/cpp_pointer_operators.htm) is pointer to a variable. For example \*var; will pointer to a variable var. |

## **Ưu tiên toán tử trong C++**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_operators_precedence.htm)

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

# LOOP TYPES

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C++ programming language provides the following type of loops to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No** | **Loop Type & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/cplusplus/cpp_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/cplusplus/cpp_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [do...while loop](https://www.tutorialspoint.com/cplusplus/cpp_do_while_loop.htm)  Like a ‘while’ statement, except that it tests the condition at the end of the loop body. |
| 4 | [nested loops](https://www.tutorialspoint.com/cplusplus/cpp_nested_loops.htm)  You can use one or more loop inside any another ‘while’, ‘for’ or ‘do..while’ loop. |

## **Loop Control Statements**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

|  |  |
| --- | --- |
| **Sr.No** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/cplusplus/cpp_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [continue statement](https://www.tutorialspoint.com/cplusplus/cpp_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [goto statement](https://www.tutorialspoint.com/cplusplus/cpp_goto_statement.htm)  Transfers control to the labeled statement. Though it is not advised to use goto statement in your program. |

## **The Infinite Loop**

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty.

#include <iostream>

using namespace std;

int main () {

for( ; ; ) {

printf("This loop will run forever.\n");

}

return 0;

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the ‘for (;;)’ construct to signify an infinite loop.

**NOTE** − You can terminate an infinite loop by pressing Ctrl + C keys.

# DECISION MAKING

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C++ programming language provides following types of decision making statements.

|  |  |
| --- | --- |
| **Sr.No** | **Statement & Description** |
| 1 | [if statement](https://www.tutorialspoint.com/cplusplus/cpp_if_statement.htm)  An ‘if’ statement consists of a boolean expression followed by one or more statements. |
| 2 | [if...else statement](https://www.tutorialspoint.com/cplusplus/cpp_if_else_statement.htm)  An ‘if’ statement can be followed by an optional ‘else’ statement, which executes when the boolean expression is false. |
| 3 | [switch statement](https://www.tutorialspoint.com/cplusplus/cpp_switch_statement.htm)  A ‘switch’ statement allows a variable to be tested for equality against a list of values. |
| 4 | [nested if statements](https://www.tutorialspoint.com/cplusplus/cpp_nested_if.htm)  You can use one ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement(s). |
| 5 | [nested switch statements](https://www.tutorialspoint.com/cplusplus/cpp_nested_switch.htm)  You can use one ‘switch’ statement inside another ‘switch’ statement(s). |

## **The ? : Operator**

We have covered [conditional operator “? :”](https://www.tutorialspoint.com/cplusplus/cpp_conditional_operator.htm) in previous chapter which can be used to replace **if...else** statements. It has the following general form −

Exp1 ? Exp2 : Exp3;

Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ‘?’ expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ‘?’ expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

# FUNCTIONS

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

## **Defining a Function**

The general form of a C++ function definition is as follows −

return\_type function\_name( parameter list ) {

body of the function

}

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

* **Return Type** − A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body** − The function body contains a collection of statements that define what the function does.

## **Example**

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

// function returning the max between two numbers

int max(int num1, int num2) {

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

## **Function Declarations**

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return\_type function\_name( parameter list );

For the above defined function max(), following is the function declaration −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

## **Calling a Function**

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it’s return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

[Live Demo](http://tpcg.io/XuXfLz)

#include <iostream>

using namespace std;

// function declaration

int max(int num1, int num2);

int main () {

// local variable declaration:

int a = 100;

int b = 200;

int ret;

// calling a function to get max value.

ret = max(a, b);

cout << "Max value is : " << ret << endl;

return 0;

}

// function returning the max between two numbers

int max(int num1, int num2) {

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

Max value is : 200

## **Function Arguments**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function −

|  |  |
| --- | --- |
| **Sr.No** | **Call Type & Description** |
| 1 | [Call by Value](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_value.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [Call by Pointer](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_pointer.htm)  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |
| 3 | [Call by Reference](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_reference.htm)  This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

## **Default Values for Parameters**

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

[Live Demo](http://tpcg.io/KoJLUN)

#include <iostream>

using namespace std;

int sum(int a, int b = 20) {

int result;

result = a + b;

return (result);

}

int main () {

// local variable declaration:

int a = 100;

int b = 200;

int result;

// calling a function to add the values.

result = sum(a, b);

cout << "Total value is :" << result << endl;

// calling a function again as follows.

result = sum(a);

cout << "Total value is :" << result << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total value is :300

Total value is :120

# NUMBERS

Normally, when we work with Numbers, we use primitive data types such as int, short, long, float and double, etc. The number data types, their possible values and number ranges have been explained while discussing C++ Data Types.

## **Defining Numbers in C++**

You have already defined numbers in various examples given in previous chapters. Here is another consolidated example to define various types of numbers in C++ −

[Live Demo](http://tpcg.io/Q21eQX)

#include <iostream>

using namespace std;

int main () {

// number definition:

short s;

int i;

long l;

float f;

double d;

// number assignments;

s = 10;

i = 1000;

l = 1000000;

f = 230.47;

d = 30949.374;

// number printing;

cout << "short s :" << s << endl;

cout << "int i :" << i << endl;

cout << "long l :" << l << endl;

cout << "float f :" << f << endl;

cout << "double d :" << d << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

short s :10

int i :1000

long l :1000000

float f :230.47

double d :30949.4

## **Math Operations in C++**

In addition to the various functions you can create, C++ also includes some useful functions you can use. These functions are available in standard C and C++ libraries and called **built-in** functions. These are functions that can be included in your program and then use.

C++ has a rich set of mathematical operations, which can be performed on various numbers. Following table lists down some useful built-in mathematical functions available in C++.

To utilize these functions you need to include the math header file **<cmath>**.

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| 1 | **double cos(double);**  This function takes an angle (as a double) and returns the cosine. |
| 2 | **double sin(double);**  This function takes an angle (as a double) and returns the sine. |
| 3 | **double tan(double);**  This function takes an angle (as a double) and returns the tangent. |
| 4 | **double log(double);**  This function takes a number and returns the natural log of that number. |
| 5 | **double pow(double, double);**  The first is a number you wish to raise and the second is the power you wish to raise it t |
| 6 | **double hypot(double, double);**  If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse. |
| 7 | **double sqrt(double);**  You pass this function a number and it gives you the square root. |
| 8 | **int abs(int);**  This function returns the absolute value of an integer that is passed to it. |
| 9 | **double fabs(double);**  This function returns the absolute value of any decimal number passed to it. |
| 10 | **double floor(double);**  Finds the integer which is less than or equal to the argument passed to it. |

Following is a simple example to show few of the mathematical operations −

[Live Demo](http://tpcg.io/qU7JSv)

#include <iostream>

#include <cmath>

using namespace std;

int main () {

// number definition:

short s = 10;

int i = -1000;

long l = 100000;

float f = 230.47;

double d = 200.374;

// mathematical operations;

cout << "sin(d) :" << sin(d) << endl;

cout << "abs(i) :" << abs(i) << endl;

cout << "floor(d) :" << floor(d) << endl;

cout << "sqrt(f) :" << sqrt(f) << endl;

cout << "pow( d, 2) :" << pow(d, 2) << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

sign(d) :-0.634939

abs(i) :1000

floor(d) :200

sqrt(f) :15.1812

pow( d, 2 ) :40149.7

## **Random Numbers in C++**

There are many cases where you will wish to generate a random number. There are actually two functions you will need to know about random number generation. The first is **rand()**, this function will only return a pseudo random number. The way to fix this is to first call the **srand()** function.

Following is a simple example to generate few random numbers. This example makes use of **time()** function to get the number of seconds on your system time, to randomly seed the rand() function −

[Live Demo](http://tpcg.io/Z1ZN0e)

#include <iostream>

#include <ctime>

#include <cstdlib>

using namespace std;

int main () {

int i,j;

// set the seed

srand( (unsigned)time( NULL ) );

/\* generate 10 random numbers. \*/

for( i = 0; i < 10; i++ ) {

// generate actual random number

j = rand();

cout <<" Random Number : " << j << endl;

}

return 0;

}

When the above code is compiled and executed, it produces the following result –

Random Number : 1748144778

Random Number : 630873888

Random Number : 2134540646

Random Number : 219404170

Random Number : 902129458

Random Number : 920445370

Random Number : 1319072661

Random Number : 257938873

Random Number : 1256201101

Random Number : 580322989

# ARRAYS

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

## **Declaring Arrays**

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[10];

## **Initializing Arrays**

You can initialize C++ array elements either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

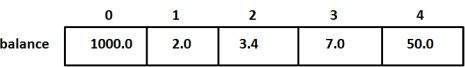
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above −



## **Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

[Live Demo](http://tpcg.io/QRe4fY)

#include <iostream>

using namespace std;

#include <iomanip>

using std::setw;

int main () {

int n[ 10 ]; // n is an array of 10 integers

// initialize elements of array n to 0

for ( int i = 0; i < 10; i++ ) {

n[ i ] = i + 100; // set element at location i to i + 100

}

cout << "Element" << setw( 13 ) << "Value" << endl;

// output each array element's value

for ( int j = 0; j < 10; j++ ) {

cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;

}

return 0;

}

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result −

Element Value

0 100

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

## **Arrays in C++**

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [Multi-dimensional arrays](https://www.tutorialspoint.com/cplusplus/cpp_multi_dimensional_arrays.htm)  C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| 2 | [Pointer to an array](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_an_array.htm)  You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |
| 3 | [Passing arrays to functions](https://www.tutorialspoint.com/cplusplus/cpp_passing_arrays_to_functions.htm)  You can pass to the function a pointer to an array by specifying the array's name without an index. |
| 4 | [Return array from functions](https://www.tutorialspoint.com/cplusplus/cpp_return_arrays_from_functions.htm)  C++ allows a function to return an array. |

# STRINGS

C++ provides following two types of string representations −

* The C-style character string.
* The string class type introduced with Standard C++.

## **The C-Style Character String**

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

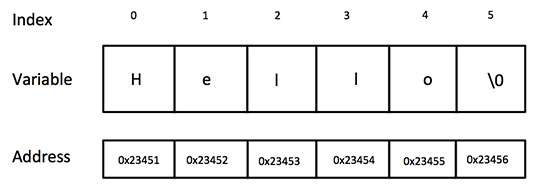
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization, then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

[Live Demo](http://tpcg.io/OzjL2L)

#include <iostream>

using namespace std;

int main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

cout << "Greeting message: ";

cout << greeting << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Greeting message: Hello

C++ supports a wide range of functions that manipulate null-terminated strings −

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| 1 | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**  Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);**  Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);**  Returns a pointer to the first occurrence of string s2 in string s1. |

Following example makes use of few of the above-mentioned functions −

[Live Demo](http://tpcg.io/oA6mP3)

#include <iostream>

#include <cstring>

using namespace std;

int main () {

char str1[10] = "Hello";

char str2[10] = "World";

char str3[10];

int len ;

// copy str1 into str3

strcpy( str3, str1);

cout << "strcpy( str3, str1) : " << str3 << endl;

// concatenates str1 and str2

strcat( str1, str2);

cout << "strcat( str1, str2): " << str1 << endl;

// total lenghth of str1 after concatenation

len = strlen(str1);

cout << "strlen(str1) : " << len << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10

## **The String Class in C++**

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

[Live Demo](http://tpcg.io/upuWFC)

#include <iostream>

#include <string>

using namespace std;

int main () {

string str1 = "Hello";

string str2 = "World";

string str3;

int len ;

// copy str1 into str3

str3 = str1;

cout << "str3 : " << str3 << endl;

// concatenates str1 and str2

str3 = str1 + str2;

cout << "str1 + str2 : " << str3 << endl;

// total length of str3 after concatenation

len = str3.size();

cout << "str3.size() : " << len << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

str3 : Hello

str1 + str2 : HelloWorld

str3.size() : 10

# CON TRỎ

Con trỏ C++ rất dễ học và thú vị. Một số tác vụ C++ được thực hiện dễ dàng hơn với con trỏ và các tác vụ C++ khác, chẳng hạn như cấp phát bộ nhớ động, không thể thực hiện được nếu không có chúng.

Như bạn đã biết, mọi biến là một vị trí bộ nhớ và mọi vị trí bộ nhớ đều có địa chỉ được xác định có thể được truy cập bằng toán tử dấu và (&) biểu thị một địa chỉ trong bộ nhớ. Hãy xem xét những điều sau đây sẽ in địa chỉ của các biến được xác định:

#include <iostream>

using namespace std;

int main () {

int var1;

char var2[10];

cout << "Address of var1 variable: ";

cout << &var1 << endl;

cout << "Address of var2 variable: ";

cout << &var2 << endl;

return 0;

}

Kết quả :

Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6

## **Con trỏ là gì?**

Con trỏ là một biến có giá trị là địa chỉ của một biến khác. Giống như bất kỳ biến hay hằng nào, bạn phải khai báo một con trỏ trước khi có thể làm việc với nó. Dạng tổng quát của một khai báo biến con trỏ là:

type \*var-name;

Ở đây, loại là loại cơ sở của con trỏ; nó phải là một kiểu C++ hợp lệ và var-name là tên của biến con trỏ. Dấu hoa thị mà bạn đã sử dụng để khai báo một con trỏ cũng chính là dấu hoa thị mà bạn sử dụng để nhân. Tuy nhiên, trong câu lệnh này, dấu hoa thị đang được sử dụng để chỉ định một biến làm con trỏ. Sau đây là khai báo con trỏ hợp lệ :

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

float \*fp; // pointer to a float

char \*ch // pointer to character

Kiểu dữ liệu thực tế của giá trị của tất cả các con trỏ, dù là số nguyên, số float, ký tự hay cách khác, đều giống nhau, một số thập lục phân dài đại diện cho một địa chỉ bộ nhớ. Sự khác biệt duy nhất giữa các con trỏ thuộc các kiểu dữ liệu khác nhau là kiểu dữ liệu của biến hoặc hằng mà con trỏ trỏ tới.

## **Sử dụng con trỏ trong C++**

Có một số thao tác quan trọng mà chúng ta sẽ thực hiện rất thường xuyên với con trỏ. (a) Chúng ta định nghĩa một biến con trỏ. (b) Gán địa chỉ của một biến cho một con trỏ. (c) Cuối cùng truy cập giá trị tại địa chỉ có sẵn trong biến con trỏ. Điều này được thực hiện bằng cách sử dụng toán tử một ngôi \* trả về giá trị của biến nằm tại địa chỉ được chỉ định bởi toán hạng của nó. Ví dụ sau sử dụng các thao tác này:

#include <iostream>

using namespace std;

int main () {

int var = 20; // actual variable declaration.

int \*ip; // pointer variable

ip = &var; // store address of var in pointer variable

cout << "Value of var variable: ";

cout << var << endl;

// print the address stored in ip pointer variable

cout << "Address stored in ip variable: ";

cout << ip << endl;

// access the value at the address available in pointer

cout << "Value of \*ip variable: ";

cout << \*ip << endl;

return 0;

}

Kết quả :

Value of var variable: 20

Address stored in ip variable: 0xbfc601ac

Value of \*ip variable: 20

## **Con trỏ trong C++**

Con trỏ có nhiều khái niệm nhưng dễ hiểu và chúng rất quan trọng đối với lập trình C++. Có một số khái niệm con trỏ quan trọng sau đây cần phải rõ ràng đối với một lập trình viên C++ :

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [Null Pointers](https://www.tutorialspoint.com/cplusplus/cpp_null_pointers.htm)  C++ hỗ trợ con trỏ null, là hằng số có giá trị bằng 0 được xác định trong một số thư viện chuẩn. |
| 2 | [Pointer Arithmetic](https://www.tutorialspoint.com/cplusplus/cpp_pointer_arithmatic.htm)  Có bốn toán tử số học có thể được sử dụng trên con trỏ: ++, --, +, - |
| 3 | [Pointers vs Arrays](https://www.tutorialspoint.com/cplusplus/cpp_pointers_vs_arrays.htm)  There is a close relationship between pointers and arrays. |
| 4 | [Array of Pointers](https://www.tutorialspoint.com/cplusplus/cpp_array_of_pointers.htm)  You can define arrays to hold a number of pointers. |
| 5 | [Pointer to Pointer](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_pointer.htm)  C++ allows you to have pointer on a pointer and so on. |
| 6 | [Passing Pointers to Functions](https://www.tutorialspoint.com/cplusplus/cpp_passing_pointers_to_functions.htm)  Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function. |
| 7 | [Return Pointer from Functions](https://www.tutorialspoint.com/cplusplus/cpp_return_pointer_from_functions.htm)  C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well. |

# REFERENCES

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

## **References vs Pointers**

References are often confused with pointers but three major differences between references and pointers are −

* You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
* Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
* A reference must be initialized when it is created. Pointers can be initialized at any time.

## **Creating References in C++**

Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. For example, suppose we have the following example −

int i = 17;

We can declare reference variables for i as follows.

int& r = i;

Read the & in these declarations as **reference**. Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration as "s is a double reference initialized to d.". Following example makes use of references on int and double −

[Live Demo](http://tpcg.io/SS8zU0)

#include <iostream>

using namespace std;

int main () {

// declare simple variables

int i;

double d;

// declare reference variables

int& r = i;

double& s = d;

i = 5;

cout << "Value of i : " << i << endl;

cout << "Value of i reference : " << r << endl;

d = 11.7;

cout << "Value of d : " << d << endl;

cout << "Value of d reference : " << s << endl;

return 0;

}

When the above code is compiled together and executed, it produces the following result −

Value of i : 5

Value of i reference : 5

Value of d : 11.7

Value of d reference : 11.7

References are usually used for function argument lists and function return values. So following are two important subjects related to C++ references which should be clear to a C++ programmer −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [References as Parameters](https://www.tutorialspoint.com/cplusplus/passing_parameters_by_references.htm)  C++ supports passing references as function parameter more safely than parameters. |
| 2 | [Reference as Return Value](https://www.tutorialspoint.com/cplusplus/returning_values_by_reference.htm)  You can return reference from a C++ function like any other data type. |

# DATE & TIME

The C++ standard library does not provide a proper date type. C++ inherits the structs and functions for date and time manipulation from C. To access date and time related functions and structures, you would need to include <ctime> header file in your C++ program.

There are four time-related types: **clock\_t, time\_t, size\_t**, and **tm**. The types - clock\_t, size\_t and time\_t are capable of representing the system time and date as some sort of integer.

The structure type **tm** holds the date and time in the form of a C structure having the following elements −

struct tm {

int tm\_sec; // seconds of minutes from 0 to 61

int tm\_min; // minutes of hour from 0 to 59

int tm\_hour; // hours of day from 0 to 24

int tm\_mday; // day of month from 1 to 31

int tm\_mon; // month of year from 0 to 11

int tm\_year; // year since 1900

int tm\_wday; // days since sunday

int tm\_yday; // days since January 1st

int tm\_isdst; // hours of daylight savings time

}

Following are the important functions, which we use while working with date and time in C or C++. All these functions are part of standard C and C++ library and you can check their detail using reference to C++ standard library given below.

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| 1 | **time\_t time(time\_t \*time);**  This returns the current calendar time of the system in number of seconds elapsed since January 1, 1970. If the system has no time, .1 is returned. |
| 2 | **char \*ctime(const time\_t \*time);**  This returns a pointer to a string of the form *day month year hours:minutes:seconds year\n\0*. |
| 3 | **struct tm \*localtime(const time\_t \*time);**  This returns a pointer to the **tm** structure representing local time. |
| 4 | **clock\_t clock(void);**  This returns a value that approximates the amount of time the calling program has been running. A value of .1 is returned if the time is not available. |
| 5 | **char \* asctime ( const struct tm \* time );**  This returns a pointer to a string that contains the information stored in the structure pointed to by time converted into the form: day month date hours:minutes:seconds year\n\0 |
| 6 | **struct tm \*gmtime(const time\_t \*time);**  This returns a pointer to the time in the form of a tm structure. The time is represented in Coordinated Universal Time (UTC), which is essentially Greenwich Mean Time (GMT). |
| 7 | **time\_t mktime(struct tm \*time);**  This returns the calendar-time equivalent of the time found in the structure pointed to by time. |
| 8 | **double difftime ( time\_t time2, time\_t time1 );**  This function calculates the difference in seconds between time1 and time2. |
| 9 | **size\_t strftime();**  This function can be used to format date and time in a specific format. |

## **Current Date and Time**

Suppose you want to retrieve the current system date and time, either as a local time or as a Coordinated Universal Time (UTC). Following is the example to achieve the same −

[Live Demo](http://tpcg.io/DHKMA9)

#include <iostream>

#include <ctime>

using namespace std;

int main() {

// current date/time based on current system

time\_t now = time(0);

// convert now to string form

char\* dt = ctime(&now);

cout << "The local date and time is: " << dt << endl;

// convert now to tm struct for UTC

tm \*gmtm = gmtime(&now);

dt = asctime(gmtm);

cout << "The UTC date and time is:"<< dt << endl;

}

When the above code is compiled and executed, it produces the following result −

The local date and time is: Sat Jan 8 20:07:41 2011

The UTC date and time is:Sun Jan 9 03:07:41 2011

## **Format Time using struct tm**

The **tm** structure is very important while working with date and time in either C or C++. This structure holds the date and time in the form of a C structure as mentioned above. Most of the time related functions makes use of tm structure. Following is an example which makes use of various date and time related functions and tm structure −

While using structure in this chapter, I'm making an assumption that you have basic understanding on C structure and how to access structure members using arrow -> operator.

[Live Demo](http://tpcg.io/SMnO0N)

#include <iostream>

#include <ctime>

using namespace std;

int main() {

// current date/time based on current system

time\_t now = time(0);

cout << "Number of sec since January 1,1970 is:: " << now << endl;

tm \*ltm = localtime(&now);

// print various components of tm structure.

cout << "Year:" << 1900 + ltm->tm\_year<<endl;

cout << "Month: "<< 1 + ltm->tm\_mon<< endl;

cout << "Day: "<< ltm->tm\_mday << endl;

cout << "Time: "<< 5+ltm->tm\_hour << ":";

cout << 30+ltm->tm\_min << ":";

cout << ltm->tm\_sec << endl;

}

When the above code is compiled and executed, it produces the following result −

Number of sec since January 1,1970 is:: 1588485717

Year:2020

Month: 5

Day: 3

Time: 11:31:57

# BASIC INPUT / INPUT

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

## **I/O Library Header Files**

There are following header files important to C++ programs −

|  |  |
| --- | --- |
| **Sr.No** | **Header File & Function and Description** |
| 1 | **<iostream>**  This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively. |
| 2 | **<iomanip>**  This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**. |
| 3 | **<fstream>**  This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter. |

## **The Standard Output Stream (cout)**

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

[Live Demo](http://tpcg.io/qFaiKd)

#include <iostream>

using namespace std;

int main() {

char str[] = "Hello C++";

cout << "Value of str is : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Value of str is : Hello C++

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

## **The Standard Input Stream (cin)**

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

[Live Demo](http://tpcg.io/NDGHmy)

#include <iostream>

using namespace std;

int main() {

char name[50];

cout << "Please enter your name: ";

cin >> name;

cout << "Your name is: " << name << endl;

}

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

Please enter your name: cplusplus

Your name is: cplusplus

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

cin >> name >> age;

This will be equivalent to the following two statements −

cin >> name;

cin >> age;

## **The Standard Error Stream (cerr)**

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

[Live Demo](http://tpcg.io/3xp5lM)

#include <iostream>

using namespace std;

int main() {

char str[] = "Unable to read....";

cerr << "Error message : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Error message : Unable to read....

## **The Standard Log Stream (clog)**

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

[Live Demo](http://tpcg.io/Dewejb)

#include <iostream>

using namespace std;

int main() {

char str[] = "Unable to read....";

clog << "Error message : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Error message : Unable to read....

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

# DATA STRUCTES

C/C++ arrays allow you to define variables that combine several data items of the same kind, but **structure** is another user defined data type which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

## **Defining a Structure**

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this −

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} book;

## **Accessing Structure Members**

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure −

[Live Demo](http://tpcg.io/zvIEi0)

#include <iostream>

#include <cstring>

using namespace std;

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main() {

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// book 1 specification

strcpy( Book1.title, "Learn C++ Programming");

strcpy( Book1.author, "Chand Miyan");

strcpy( Book1.subject, "C++ Programming");

Book1.book\_id = 6495407;

// book 2 specification

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Yakit Singha");

strcpy( Book2.subject, "Telecom");

Book2.book\_id = 6495700;

// Print Book1 info

cout << "Book 1 title : " << Book1.title <<endl;

cout << "Book 1 author : " << Book1.author <<endl;

cout << "Book 1 subject : " << Book1.subject <<endl;

cout << "Book 1 id : " << Book1.book\_id <<endl;

// Print Book2 info

cout << "Book 2 title : " << Book2.title <<endl;

cout << "Book 2 author : " << Book2.author <<endl;

cout << "Book 2 subject : " << Book2.subject <<endl;

cout << "Book 2 id : " << Book2.book\_id <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Book 1 title : Learn C++ Programming

Book 1 author : Chand Miyan

Book 1 subject : C++ Programming

Book 1 id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Yakit Singha

Book 2 subject : Telecom

Book 2 id : 6495700

## **Structures as Function Arguments**

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example −

[Live Demo](http://tpcg.io/xvb7BQ)

#include <iostream>

#include <cstring>

using namespace std;

void printBook( struct Books book );

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main() {

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// book 1 specification

strcpy( Book1.title, "Learn C++ Programming");

strcpy( Book1.author, "Chand Miyan");

strcpy( Book1.subject, "C++ Programming");

Book1.book\_id = 6495407;

// book 2 specification

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Yakit Singha");

strcpy( Book2.subject, "Telecom");

Book2.book\_id = 6495700;

// Print Book1 info

printBook( Book1 );

// Print Book2 info

printBook( Book2 );

return 0;

}

void printBook( struct Books book ) {

cout << "Book title : " << book.title <<endl;

cout << "Book author : " << book.author <<endl;

cout << "Book subject : " << book.subject <<endl;

cout << "Book id : " << book.book\_id <<endl;

}

When the above code is compiled and executed, it produces the following result −

Book title : Learn C++ Programming

Book author : Chand Miyan

Book subject : C++ Programming

Book id : 6495407

Book title : Telecom Billing

Book author : Yakit Singha

Book subject : Telecom

Book id : 6495700

## **Pointers to Structures**

You can define pointers to structures in very similar way as you define pointer to any other variable as follows −

struct Books \*struct\_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows −

struct\_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows −

struct\_pointer->title;

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept −

[Live Demo](http://tpcg.io/P8IeUE)

#include <iostream>

#include <cstring>

using namespace std;

void printBook( struct Books \*book );

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main() {

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// Book 1 specification

strcpy( Book1.title, "Learn C++ Programming");

strcpy( Book1.author, "Chand Miyan");

strcpy( Book1.subject, "C++ Programming");

Book1.book\_id = 6495407;

// Book 2 specification

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Yakit Singha");

strcpy( Book2.subject, "Telecom");

Book2.book\_id = 6495700;

// Print Book1 info, passing address of structure

printBook( &Book1 );

// Print Book1 info, passing address of structure

printBook( &Book2 );

return 0;

}

// This function accept pointer to structure as parameter.

void printBook( struct Books \*book ) {

cout << "Book title : " << book->title <<endl;

cout << "Book author : " << book->author <<endl;

cout << "Book subject : " << book->subject <<endl;

cout << "Book id : " << book->book\_id <<endl;

}

When the above code is compiled and executed, it produces the following result −

Book title : Learn C++ Programming

Book author : Chand Miyan

Book subject : C++ Programming

Book id : 6495407

Book title : Telecom Billing

Book author : Yakit Singha

Book subject : Telecom

Book id : 6495700

## **The typedef Keyword**

There is an easier way to define structs or you could "alias" types you create. For example −

typedef struct {

char title[50];

char author[50];

char subject[100];

int book\_id;

} Books;

Now, you can use *Books* directly to define variables of *Books* type without using struct keyword. Following is the example −

Books Book1, Book2;

You can use **typedef** keyword for non-structs as well as follows −

typedef long int \*pint32;

pint32 x, y, z;

x, y and z are all pointers to long ints.