1. **Kiểu dữ liệu trong C++**
   1. **Kiểu dữ liệu nguyên thủy trong C/C++**

* Tên tiếng Anh là Primitive Type, còn có thể gọi là kiểu dữ liệu gốc, kiểu dữ liệu có sẵn trong C/C++. Bên cạnh các kiểu dữ liệu gốc này, C/C++ cũng cung cấp các kiểu dữ liệu user-defined. Bảng dưới đây liệt kê 7 kiểu dữ liệu cơ bản trong C/C++:

|  |  |
| --- | --- |
| **Kiểu dữ liệu** | **Từ khóa** |
| Boolean | bool |
| Ký tự | char |
| Số nguyên | int |
| Số thực | float |
| Số thực dạng Double | double |
| Kiểu không có giá trị | void |
| Kiểu Wide character | wchar\_t |

* Một số kiểu cơ bản có thể được sửa đổi bởi sử dụng một hoặc nhiều modifier này:
  + Signed (kiểu có dấu)
  + Unsigned (kiểu không có dấu)
  + Short
  + Long
* Bảng sau hiển thị kiểu biến, lượng bộ nhớ nó dùng để lưu giá trị trong bộ nhớ, và giá trị lớn nhất và nhỏ nhất có thể được lưu giữ với các kiểu biến đó:

|  |  |  |
| --- | --- | --- |
| **Kiểu** | **Độ rộng bit** | **Dãy giá trị** |
| char | 1 byte | -127 tới 127 hoặc 0 tới 255 |
| unsigned char | 1 byte | 0 tới 255 |
| signed char | 1 byte | -127 tới 127 |
| int | 4 byte | -2147483648 tới 2147483647 |
| unsigned int | 4 byte | 0 tới 4294967295 |
| signed int | 4 byte | -2147483648 tới 2147483647 |
| short int | 2 byte | -32768 tới 32767 |
| unsigned short int | Range | 0 tới 65,535 |
| signed short int | Range | -32768 tới 32767 |
| long int | 4 byte | -2,147,483,647 tới 2,147,483,647 |
| signed long int | 4 byte | Tương tự như long int |
| unsigned long int | 4 byte | 0 tới 4,294,967,295 |
| float | 4 byte | +/- 3.4e +/- 38 (~7 chữ số) |
| double | 8 byte | +/- 1.7e +/- 308 (~15 chữ số) |
| long double | 8 byte | +/- 1.7e +/- 308 (~15 chữ số) |
| wchar\_t | 2 hoặc 4 byte | 1 wide character |

* Kích cỡ của các biến có thể khác với những gì hiển thị trên bảng, phụ thuộc vào compiler và máy tính bạn đang sử dụng.
* Dưới đây là ví dụ sẽ đưa ra kích cỡ chính xác của các kiểu dữ liệu đa dạng trên máy tính của bạn.

#include <iostream>

using namespace std;

int main()

{

cout << "Kich co cua char la: " << sizeof(char) << endl;

cout << "Kich co cua int la: " << sizeof(int) << endl;

cout << "Kich co cua short int la: " << sizeof(short int) << endl;

cout << "Kich co cua long int la: " << sizeof(long int) << endl;

cout << "Kich co cua float la: " << sizeof(float) << endl;

cout << "Kich co cua double la: " << sizeof(double) << endl;

cout << "Kich co cua wchar\_t la: " << sizeof(wchar\_t) << endl;

return 0;

}

* Toán tử **sizeof()** để lấy kích cỡ của các kiểu dữ liệu đa dạng.

Kich co cua char la : 1

Kich co cua int la : 4

Kich co cua short int la : 2

Kich co cua long int la : 4

Kich co cua float la : 4

Kich co cua double la : 8

Kich co cua wchar\_t la : 4

* 1. **Khai báo typedef trong C/C++**
* Tạo một tên mới cho một kiểu dữ liệu đang tồn tại bởi sử dụng **typedef** trong C/C++. Cú pháp đơn giản sau để định nghĩa một kiểu dữ liệu mới bởi sử dụng typedef:

typedef kieu\_du\_lieu ten\_moi;

* Ví dụ sau nói cho compiler rằng sothuc là tên khác của float:

typedef float sothuc;

* Bây giờ, khai báo sau là hoàn toàn hợp lệ và sẽ tạo một biến số thực gọi là *vantoc*:

sothuc vantoc;

* 1. **Kiểu liệt kê enum trong C/C++**
* Khai báo một tên kiểu tùy ý và một tập hợp của 0 hoặc nhiều Identifier (Định danh) mà có thể được sử dụng như là các giá trị của kiểu đó. Mỗi Enumerator là một constant có kiểu là kiểu liệt kê (enumeration).
* Để tạo một Enumeration, bạn sử dụng từ khóa **enum** trong C/C++. Form chung của kiểu liệt kê enum là:

enum ten\_cua\_enum { danh sach cac ten } danh\_sach\_bien;

* Tại đây, ten\_cua\_enum là tên kiểu liệt kê. Danh sách tên được phân biệt bởi dấu phảy.
* Ví dụ, code sau định nghĩa một tên kiểu liệt kê hàng hóa gọi là *hanghoa* và biến c là kiểu của hanghoa. Cuối cùng, c được gán giá trị nuocngot.

enum hanghoa { sua, nuocngot, biachai } c;

c = nuocngot;

* Mặc định, trong danh sách các tên thì giá trị của tên đầu tiên là 0, tên thứ hai là 1 và tên thứ 3 là 2, …
* Nhưng, bạn có thể cung cấp cho một tên một giá trị cụ thể bằng việc thêm một Initializer (giá trị khởi tạo). Ví dụ, trong enumeration sau, **nuocngot** sẽ có giá trị là 40:

enum hanghoa { sua, nuocngot = 40, biachai };

1. **Phạm vi biến trong C++**

* Scope (phạm vi) là nơi **biến** hoạt động, và nói chung có thể có 3 khu vực mà biến có thể được khai báo:
  + Biến cục bộ (local): Bên trong một hàm hoặc một khối.
  + Các tham số chính thức (formal): Trong định nghĩa của các tham số hàm.
  + Biến toàn cục (global): Bên ngoài của tất cả hàm.
  1. **Biến cục bộ trong C++**
* Các biến được khai báo bên trong một hàm hoặc khối.
* Chỉ có thể được sử dụng bởi các lệnh bên trong hàm hoặc khối code đó.
* Các biến cục bộ chỉ được sử dụng bên trong hàm hoặc khối code đó.
* Ví dụ:

#include <iostream>

using namespace std;

int main()

{

// phan khai bao bien cuc bo:

int a, b;

int c;

// phan khoi tao bien

a = 10;

b = 20;

c = a + b;

cout << c;

return 0;

}

* 1. **Biến toàn cục trong C++**
* Định nghĩa bên ngoài các hàm, thường ở phần đầu chương trình.
* Các biến toàn cục giữ giá trị của nó trong suốt vòng đời chương trình của bạn.
* Có thể được truy cập bởi bất kỳ hàm nào.
* Ví dụ sử dụng biến toàn cục và biến nội bộ trong C++:

#include <iostream>

using namespace std;

// phan khai bao bien toan cuc:

int g;

int main()

{

// phan khai bao bien cuc bo:

int a, b;

// phan khoi tao bien

a = 10;

b = 20;

g = a + b;

cout << g;

return 0;

}

* Biến toàn cục và biến cục bộ cùng tên với nhau. Nhưng, trong một hàm thì giá trị của biến cục bộ sẽ được ưu tiên. Ví dụ:

#include <iostream>

using namespace std;

// phan khai bao bien toan cuc:

int g = 20;

int main()

{

// phan khai bao bien cuc bo:

int g = 10;

cout << g;

return 0;

}

* Khi code trên được biên dịch và thực thi, nó cho kết quả sau

10

* 1. **Khởi tạo biến cục bộ và biến toàn cục bởi hệ thống trong C++**
* Khi một biến cục bộ được định nghĩa, nó không được khởi tạo bởi hệ thống, chính bạn phải khởi tạo nó.
* Các biến toàn cục được khởi tạo tự động bởi hệ thống khi bạn định nghĩa chúng, như sau:

|  |  |
| --- | --- |
| **Kiểu dữ liệu** | **Giá trị khởi tạo** |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

1. **Hằng (Constant/ Literal) trong C/C++**

* Constant liên quan tới các giá trị cố định mà chương trình không thể thay đổi và chúng được gọi là **literals**.
* Ví dụ:

const PI = 3.14;

* Constant ở đây là PI, còn Literal là 3.14.

1. **Lớp lưu trữ (Storage Class) trong C/C++**

* Định nghĩa phạm vi và vòng đời của biến và các hàm bên trong một chương trình C++. Chúng thường đứng trước kiểu dữ liệu mà chúng tác động. Dưới đây là các lớp lưu trữ có thể được sử dụng trong C/C++:
  + Auto
  + Register
  + Static
  + Extern
  + mutable

## Lớp lưu trữ auto trong C/C++

* Lớp lưu trữ auto trong C/C++ là lớp lưu trữ mặc định cho tất cả biến cục bộ trong C/C++:

{

int diemthi;

auto int diemthi;

}

* Ví dụ trên định nghĩa hai biến với cùng lớp lưu trữ, auto chỉ có thể được sử dụng bên trong các hàm, ví dụ: cho các biến cục bộ.

## Lớp lưu trữ register trong C/C++

* Định nghĩa các biến cục bộ mà nên được lưu giữ trong một thanh ghi thay vì RAM. Nghĩa là, biến có kích cỡ tối đa bằng với kích cỡ thanh ghi (thường là 1 từ) và không thể có toán tử một ngôi '&' được áp dụng tới nó (vì không có địa chỉ bộ nhớ).

{

register int hocphi;

}

* Lớp lưu trữ register nên chỉ được dùng cho các biến yêu cầu truy cập nhanh như các biến đếm (counters). Cũng cần chú ý rằng, một biến định nghĩa với 'register' không có nghĩa là biến đó được lưu trữ trong thanh ghi. Tức là nó có thể được lưu trữ trong thanh ghi phụ thuộc vào phần cứng và giới hạn thực thi.

## Lớp lưu trữ static trong C/C++

* Lớp lưu trữ static trong C/C++ nói với compiler để giữ một biến cục bộ tồn tại trong toàn bộ thời gian sống của chương trình thay vì tạo và hủy biến mỗi lần nó vào và ra khỏi phạm vi biến. Vì vậy, các biến có static cho phép nó duy trì giá trị giữa các lần gọi hàm.
* Lớp lưu trữ static cũng có thể được áp dụng cho các biến toàn cục (global). Khi áp dụng cho biến toàn cục, nó nói với trình biên dịch rằng, phạm vi của biến toàn cục bị giới hạn trong tập tin mà nó được khai báo.
* Trong C/C++, khi static được sử dụng trên thành viên dữ liệu của lớp, nó gây ra: chỉ có một bản sao của thành viên đó được chia sẻ bởi tất cả đối tượng trong lớp của nó.

#include <iostream>

// phan khai bao ham

void func(void);

static int biendem = 10; /\* Day la bien toan cuc \*/

main()

{

while (biendem--)

{

func();

}

return 0;

}

// Phan dinh nghia ham

void func(void)

{

static int i = 5; // Day la bien cuc bo dang static

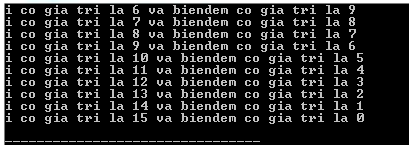
i++

std::cout << "i co gia tri la " << i;

std::cout << " va biendem co gia tri la " << biendem << std::endl;

}

* Chạy chương trình C/C++ trên sẽ cho kết quả như hình sau:



## Lớp lưu trữ extern trong C/C++

* Lớp lưu trữ **extern** trong C/C++ được dùng để cung cấp một tham chiếu của một biến toàn cục được nhìn thấy bởi TẤT CẢ các file chương trình. Khi bạn sử dụng 'extern', biến không thể được khởi tạo, khi nó trỏ tới tên biến tại một vị trí lớp lưu trữ mà đã được định nghĩa trước đó.
* Khi bạn có nhiều file và bạn định nghĩa một biến hay hàm toàn cục trong một file và cũng muốn dùng nó trong các file khác, thì **extern** được dùng trong file khác để cung cấp tham chiếu của biến hay hàm được định nghĩa. Cần nhớ rằng, **extern** dùng để khai báo một biến hay hàm toàn cục trong file khác.
* Lớp lưu trữ extern được dùng phổ biến khi có hai hoặc nhiều file chia sẻ cùng biến hay hàm toàn cục. Xem ví dụ với hai file sau:
* File đầu tiên: extern1.cpp

#include <iostream>

int biendem ;

extern void vidu\_extern();

main()

{

biendem = 5;

vidu\_extern();

}

* File thứ hai: extern2.cpp

#include <iostream>

extern int biendem;

void vidu\_extern(void)

{

std::cout << "Gia tri biendem la " << count << std::endl;

}

Ở đây, từ khóa *extern* đang được sử dụng để khai báo biendem trong file khác. Bây giờ biên dịch hai file này như sau:

$g++ extern1.cpp extern2.cpp -o write

Nó sẽ tạo chương trình **write** có thể thực thi, bạn thử thực thi write và kiểm tra kết quả như sau:

$./write

5

## Lớp lưu trữ mutable trong C/C++

Lớp lưu trữ **mutable** trong C/C++ chỉ áp dụng cho các đối tượng class, sẽ được bàn luận trong chương sau. Nó cho phép một thành viên của một đối tượng để override (ghi đè). Đó là, một thành viên là mutable có thể được sửa đổi bởi một hàm thành viên const.