Chương 8
CON TRỎ - POINTER

hiepnd@soict.hut.edu.vn

Nội dung

- Nhắc lại về tổ chức bộ nhớ của máy tính
- Biến con trỏ
- Con trỏ và cấu trúc
- Con trỏ và hàm
- Con trỏ và cấu trúc
- Con trỏ và cấp phát bộ nhớ động

Nhắc lại tổ chức bộ nhớ của máy tính

Nhắc lại về tổ chức bộ nhớ máy tính

- Trong máy tính, bộ nhớ trong :
  - chia thành các ô nhớ
  - Các ô nhớ được đánh địa chỉ khác nhau
  - Kích thước của mỗi ô nhớ là 1 byte

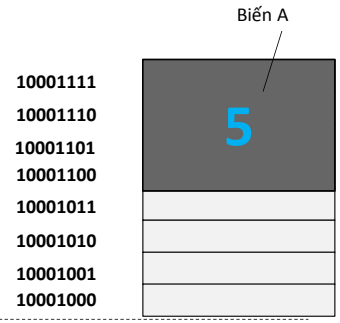
Địa chỉ ô nhớ	
11111111	10010101
11111110	11010101
11111101	10010100
	10000101
00000000	00010101

Nhắc lại về tổ chức bộ nhớ máy tính

- Khi khai báo 1 biến, các ô nhớ sẽ được cấp phát cho biến đó

  int A; // 4 byte

  A=5;
- Biến A được lưu trữ trong 4 ô bắt đầu tại địa chỉ 10001111
- Giá trị của biến A là 5 (4 ô nhớ chứa giá trị 5)
- Lấy địa chỉ ô nhớ (đầu tiên) cấp phát cho biến: dùng toán tử **&**
- &A trả về 10001111




```c
#include <stdio.h>
#include <stdlib.h> //cho ham system()

int main()
{
    int a, b;
    double c,d;
    a=5; b=7;
    c=3.5; d=10.0;
    printf("Gia tri a=%d, dia chi %#x\n",a,&a);
    printf("Gia tri b=%d, dia chi %#x\n",b,&b);
    printf("Gia tri a=%f, dia chi %#x\n",c,&c);
    printf("Gia tri a=%f, dia chi %#x\n",d,&d);
    system("pause");
    return 0;
}
```

Biến con trỏ

Biến con trỏ

- **Biến con trỏ - Pointer Variable**: giá trị của biến là một địa chỉ ô nhớ.
- Kích thước 1 biến con trỏ phụ thuộc vào các platform (môi trường ứng dụng):
  - Platform 16 bit là 2 byte.
  - Platform 32 bit là 4 byte.
  - Platform 64 bit là 8 byte.
- Khai báo biến con trỏ

KieuDuLieu *TenBien;

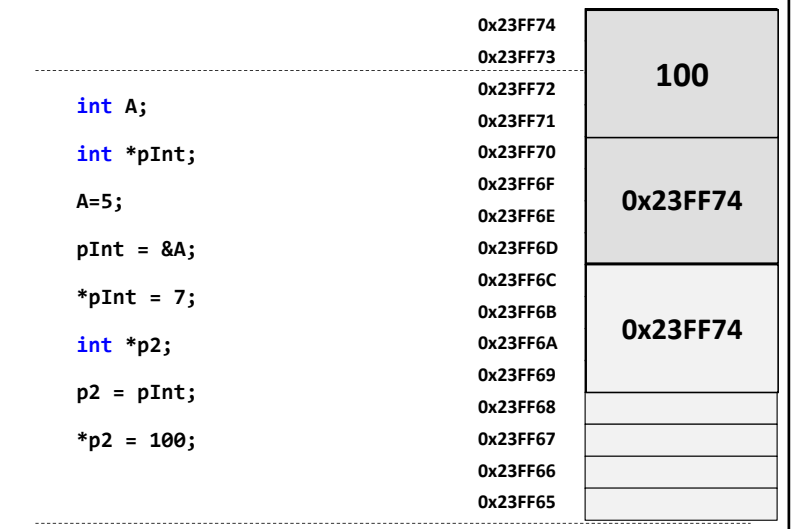
```c
int *pInt;
float *pFloat;
```

Biến con trỏ

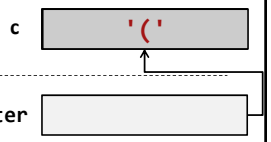
- Kích thước biến con trỏ không phụ thuộc vào kiểu dữ liệu
- Truy cập vào giá trị của vùng nhớ đang trỏ bởi con trỏ: dùng toán tử *
  *pInt là giá trị vùng nhớ trỏ bởi con trỏ pInt

```
int A=5;
int *pInt;
pInt = &A;
printf("Dia chi A = %#x, Gia tri pInt = %#x Dia chi pInt = %#x\n",
                                                    &A, pInt, &pInt);
printf("Gia tri A = %d, gia tri vung nho tro boi pInt = %d\n",A,*pInt);
*pInt = 7;
printf("Gan *pInt = 7\n");
printf("Gia tri A = %d, gia tri vung nho tro boi pInt = %d\n",A,*pInt);
```

```
int A;
int *pInt;
A=5;
pInt = &A;
*pInt = 7;
int *p2;
p2 = pInt;
*p2 = 100;
```



Biến con trỏ




```
#include <stdio.h>
int main (void)
{
  char c = 'Q';
  char *char_pointer = &c;
  printf ("%c %c\n", c, *char_pointer);
  c = '/';
  printf ("%c %c\n", c, *char_pointer);
  *char_pointer = '(';
  printf ("%c %c\n", c, *char_pointer);
  return 0;
}
```

Biến con trỏ trong biểu thức

```
#include <stdio.h>
int main (void)
{
  int i1, i2;
  int *p1, *p2;
  i1 = 5;
  p1 = &i1;
  i2 = *p1 / 2 + 10;
  p2 = p1;
  printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n",
                                    i1, i2, *p1, *p2);
  return 0;
}
```

Con trỏ hằng và hằng con trỏ

```
char c = 'X';
```

```
char *charPtr = &c;
```

Khai báo biến con trỏ thông thường

```
char * const charPtr = &c;
charPtr = &d; // not valid
```

charPtr là hằng con trỏ, nó không thể thay đổi được giá trị (không thể trỏ vào ô nhớ khác)
Có thể thay đổi giá trị của ô nhớ con trỏ đang trỏ đến

```
const char *charPtr = &c;
*charPtr = 'Y'; // not valid
```

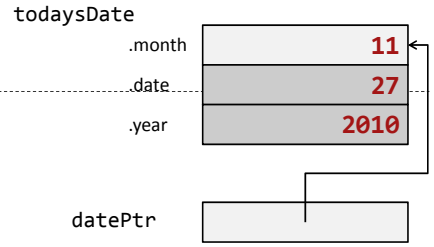
charPtr là con trỏ hằng (con trỏ tới 1 hằng số)
không thể thay đổi giá trị ô nhớ trỏ tới bởi con trỏ
(có thể cho con trỏ trỏ sang ô nhớ khác)

```
const char * const *charPtr = &c;
```

Hằng con trỏ trỏ tới hằng số: không thay đổi được cả giá trị con trỏ và giá trị ô nhớ mà nó trỏ đến

Con trỏ và cấu trúc

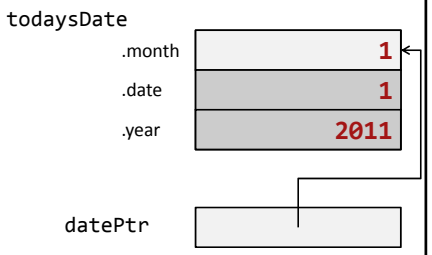
Con trỏ và cấu trúc




```
struct date
{
  int month;
  int day;
  int year;
};
struct date todaysDate ={11,27,2010};
struct date *datePtr;
datePtr = &todaysDate;
```

Con trỏ và cấu trúc

- Truy cập vào trường biến cấu trúc thông qua con trỏ
  - **(* TênConTrỏ).TênTrường**
  - **TênConTrỏ->TênTrường**




```
datePtr = &todaysDate;
datePtr->month = 1;
(*datePtr).day = 1;
datePtr->year = 2011;
```

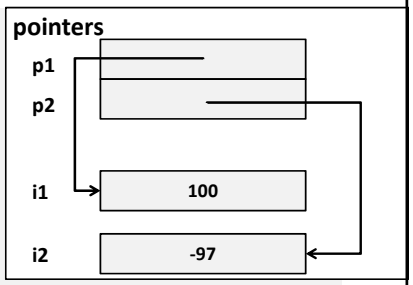
```c
#include <stdio.h>
int main (void)
{
  struct date
  {
    int month;
    int day;
    int year;
  };
  struct date today = {11,27,2010}, *datePtr;
  datePtr = &today;
  printf ("Today's date is %i/%i/%.2i.\n",datePtr->month,
                                datePtr->day, datePtr->year % 100);
  datePtr->month = 1;
  (*datePtr).day = 1;
  datePtr->year = 2011;
  printf ("Today's date is %i/%i/%.2i.\n",datePtr->month,
                                datePtr->day, datePtr->year % 100);
  return 0;
}
```

Con trỏ và cấu trúc

- Cấu trúc chứa con trỏ

```c
struct intPtrs
{
   int *p1;
   int *p2;
};
```

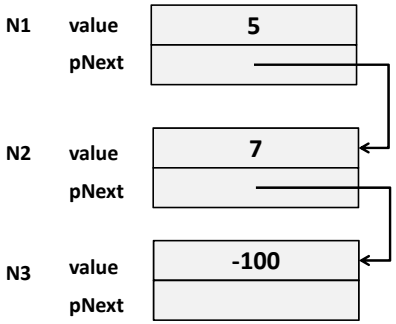
```c
#include <stdio.h>
int main (void)
{
    struct intPtrs
    {
        int *p1;
        int *p2;
    };
    struct intPtrs pointers;
    int i1 = 100, i2;
    pointers.p1 = &i1;
    pointers.p2 = &i2;
    *pointers.p2 = -97;
    printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
    printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
    return 0;
}
```



Con trỏ và cấu trúc

- **Danh sách liên kết – linked list**: một trong những cấu trúc phức tạp được xây dựng từ quan hệ con trỏ và cấu trúc

```c
struct node
{
    int value;
    struct node *pNext;
};
```




```c
#include <stdio.h>
int main (void)
{
  struct node
  {
    int value;
    struct entry *pNext;
  };
  struct node N1, N2, N3;
  int i;
  N1.value = 5; N2.value = 7; N3.value = -100;
  N1.pNext = &n2;
  N2.pNext = &n3;
  i = N1.pNext->value;
  printf ("%i ", i);
  printf ("%i\n", N2.pNext->value);
  return 0;
}
```

Con trỏ và hàm

Con trỏ và hàm

- Tham số của hàm có thể là con trỏ, và hàm có thể trả về giá trị kiểu con trỏ

```
void Absolute(int *x)
{
   if(*x<0) *x=-*x;
}

int main()
{
   int a=-6;
   printf("Before call function Absolute, a = %d\n",a);
   Absolute(&a);
   printf("After call function Absolute, a = %d\n",a);
}
```

Thay đổi giá trị vùng nhớ

Truyền vào là địa chỉ

Con trỏ và hàm

- Cách truyền tham số của hàm
  - Khi khai báo hàm, các tham số của hàm là tham số hình thức
  - Khi gọi hàm, ta truyền vào các giá trị, biến, đó là các tham số thực sự
  - Một bản copy của các tham số thực sự được gán cho các tham số hình thức của hàm, do đó mọi thay đổi giá trị trên các tham số hình thức trong khi thực hiện hàm sẽ bị mất sau khi hàm thực hiện xong (giá trị tham số thực sự không đổi)

Con trỏ và hàm

Giá trị các biến không thay đổi?

```
void Exchange(int x, int y)
{
   //exchange value of x and y, huh ?
   int tmp;
   tmp=x;
   x=y;
   y=tmp;
}
```

```
int main()
{

   int x=5, y=16;
   printf("Before function call: x = %d, y = %d\n",x,y);
   Exchange(x,y);
   printf("After function call: x = %d, y = %d\n",x,y);
   return 0;
}
```

Con trỏ và hàm

- Khi truyền tham số cho hàm là con trỏ
  - Một bản copy của con trỏ cũng được tạo ra và gán cho tham số hình thức của hàm.
  - Cả bản copy và con trỏ thực này đều cùng tham chiếu đến một vùng nhớ duy nhất, nên mọi thay đổi giá trị vùng nhớ đó (dù dùng con trỏ nào) là như nhau giống và được lưu lại
  - Sau khi kết thúc thực hiện hàm, giá trị của con trỏ không đổi, nhưng giá trị vùng nhớ mà con trỏ trỏ đến có thể được thay đổi (nếu trong hàm ta thay đổi giá trị này)
- Truyền tham số con trỏ khi ta muốn giá trị vùng nhớ được thay đổi sau khi thực hiện hàm

Con trỏ và hàm

Giá trị thực sự được thay đổi

```
void Exchange2(int *x, int *y)
{
   //it really exchange value of x and y
   int tmp;
   tmp=*x;
   *x=*y;
   *y=tmp;
}
```

```
int main()
{

   int x=5,y=16;
   printf("Before function call: x = %d, y = %d\n",x,y);
   Exchange2(&x,&y);
   printf("After function call: x = %d, y = %d\n",x,y);

   return 0;
}
```

Con trỏ và hàm

- Hàm trả về con trỏ

```
struct list
{
   int data;
   struct list *pNext;
};
```

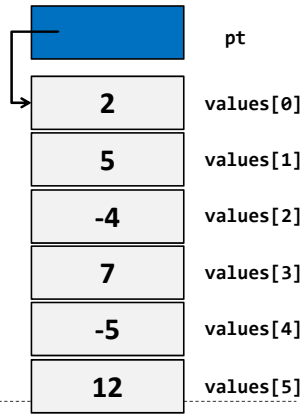
```
struct list * searchList(struct list *pHead, int key)
{
   while(pHead!=(struct list*)0) //or NULL
   {
      if(key==pHead->data)
         return pHead;
      else
         pHead=pHead->pNext;
   }
   return (struct list*)0; //or NULL
}
```

Con trỏ và mảng

Con trỏ và mảng

- Tên mảng là một con trỏ hằng trỏ vào phần tử đầu tiên của mảng

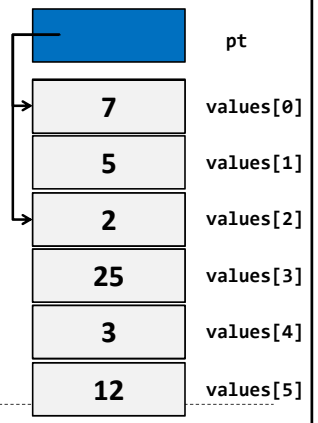
```
int values[6]={2,5,-4,7,-5,12};

int *pt;
pt = values; // same as &value[0]
```



Ta có thể dễ dàng dùng con trỏ để truy cập vào các phần tử trong mảng

Con trỏ và mảng

```
int values[6]={2,5,-4,7,-5,12};

int *pt;
pt = values; // same as &value[0]

*(pt)=7; //same as values[0]=7

*(pt+3)=25; //same as values[3]=25

pt = &values[2];//pt points to
                values[2] address

*pt = 2;//same as values[2]=2;

*(pt+2) = 3;//same as values[4]=3;
```



Con trỏ và mảng

- Một số thao tác

```
int values[6]={2,5,-4,7,-5,12};
int *pt;
```

pt = values; cho con trỏ trỏ vào phần tử đầu tiên trong mảng (chứa địa chỉ của phần tử đầu tiên)

***(pt+i)** truy cập tới **giá trị** phần tử cách phần tử đang trỏ bởi con trỏ i phần tử

pt=pt+n; //or pt+=n; cho pt trỏ tới **địa chỉ** của phần tử cách địa chỉ của phần tử hiện tại n phần tử

pt++; Cho con trỏ dịch chuyển cách 1 phần tử
pt--; (tức là sizeof(kieudulieu) ô nhớ)

Con trỏ và mảng

- Các phép toán quan hệ với con trỏ
  - Có thể sử dụng các toán tử con hệ với kiểu con trỏ
  - Các phép toán đó sẽ là so sánh các địa chỉ ô nhớ với nhau
  - Kiểu giá trị trả về là TRUE (khác 0) và FALSE (bằng 0)

```
pt>= &values[5];

pt==&values[0];
```

Con trỏ và mảng

```
int arraySum (int Array[], const int n)
{
    int sum = 0, *ptr;
    int * const arrayEnd = array + n;
    for ( ptr = Array; ptr < arrayEnd; ++ptr )
        sum += *ptr;
    return sum;
}

int main (void)
{
    int values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };
    printf ("The sum is %i\n", arraySum (values, 10));
    return 0;
}
```

Con trỏ và mảng

- Khi truyền vào mảng ta chỉ truyền địa chỉ của phần tử đầu tiên trong mảng, do đó hàm arraySum có thể viết lại là

```
int arraySum (int *Array, const int n)
{
   int sum = 0;
   int * const arrayEnd = Array + n;
   for ( ; Array < arrayEnd; ++Array )
      sum += *Array;
   return sum;
}
```

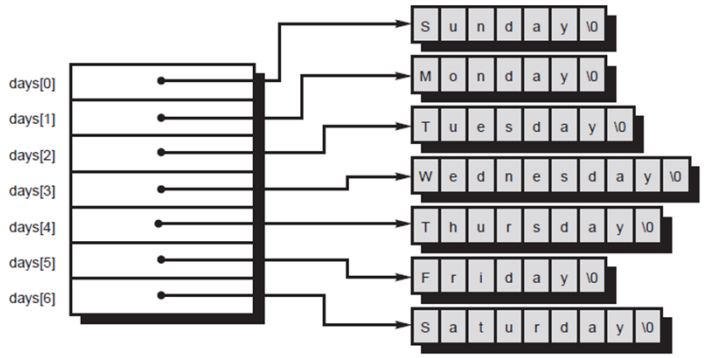
Con trỏ và mảng

```
void copyString (char to[], char from[])
{
   int i;
   for ( i = 0; from[i] != '\0'; ++i )
      to[i] = from[i];
   to[i] = '\0';
}
```

```
void copyString (char *to, char *from)
{
   for ( ; *from != '\0'; ++from, ++to )
      *to = *from;
   *to = '\0';
}
```

Con trỏ và mảng

```
char *days[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday" };
```



Con trỏ và mảng

- Sử dụng con trỏ để tìm độ dài của xâu ký tự

```
int stringLength (const char *string)
{
    const char *cptr = string;
    while ( *cptr )
       ++cptr;
    return cptr - string;
}

int main (void)
{
    printf ("%i ", stringLength ("stringLength test"));
    printf ("%i ", stringLength (""));
    printf ("%i\n", stringLength ("complete"));
    return 0;
}
```

Con trỏ và cấp phát bộ nhớ động

Con trỏ và cấp phát bộ nhớ động

- Trong nhiều trường hợp tại thời điểm lập trình ta chưa biết trước kích thước bộ nhớ cần dùng để lưu trữ dữ liệu. Ta có thể:
  - Khai báo mảng với kích thước tối đa có thể tại thời điểm biên dịch (cấp phát bộ nhớ tĩnh)
  - Sử dụng mảng với kích thước biến đổi tại thời điểm chạy (*chỉ có trong C99*)
  - Sử dụng mảng cấp phát bộ nhớ động

Con trỏ và cấp phát bộ nhớ động

- Cấp phát tĩnh
  - Kích thước bộ nhớ cấp phát được xác định ngay tại thời điểm biên dịch chương trình và không thể thay đổi trong quá trình chạy chương trình
  - Việc quản lý và thu hồi bộ nhớ được thực hiện tự động, người lập trình không cần quan tâm
  - Sẽ là rất lãng phí bộ nhớ nếu không dùng hết dung lượng được cấp
  - Bộ nhớ được lấy từ phần DATA, do đó dung lượng bộ nhớ được cấp phát tĩnh là có giới hạn

```
int A[1000];
double B[1000000]; //not enough memory
```

Con trỏ và cấp phát bộ nhớ động

- Cấp phát bộ nhớ động
  - Bộ nhớ được cấp phát tại thời điểm thực hiện chương trình, nên có thể thay đổi được trong mỗi lần chạy
  - Việc quản lý và thu hồi bộ nhớ sẽ do ***người lập trình đảm nhiệm***
  - Tiết kiệm bộ nhớ hơn so với cấp phát tĩnh (vì chỉ cần cấp phát đủ dùng)
  - Bộ nhớ cấp phát được lấy ở phần bộ nhớ rỗi (HEAP) nên dung lượng bộ nhớ có thể cấp phát lớn hơn so với cấp phát tĩnh
  - Nếu không thu hồi bộ nhớ sau khi dùng xong thì sẽ dẫn đến rò rỉ bộ nhớ (memory leak), có thể gây ra hết bộ nhớ

Con trỏ và cấp phát bộ nhớ động

- Cấp phát bộ nhớ động trong C: dùng 2 hàm malloc và calloc (trong thư viện <stdlib.h>)
  - Hàm trả về địa chỉ của ô nhớ đầu tiên trong vùng nhớ xin cấp phát, do đó dùng con trỏ để chứa địa chỉ này
  - Trả về con trỏ NULL nếu cấp phát không thành công

```
pointer=(dataType*) calloc(sizeofAnElement, noElements);
```

```
pointer = (dataType*) malloc(sizeofAnElement * noElements);
```

```
double *pt;
pt = (double *) calloc(sizeof(double),10000);
```

```
int *pInt;
pInt = (int*) malloc(sizeof(int)*10000);
```

Con trỏ và cấp phát bộ nhớ động

- Hàm calloc
  - Cần 2 tham số là kích thước 1 phần tử (theo byte) và số lượng phần tử
  - Khi cấp phát sẽ tự động đưa giá trị các ô nhớ được cấp phát về 0
- Hàm malloc
  - Chỉ cần 1 tham số là kích thước bộ nhớ (theo byte)
  - Không tự đưa giá trị các ô nhớ về 0
- Hàm sizeof
  - Trả về kích thước của 1 kiểu dữ liệu, biến (tính theo byte)

Con trỏ và cấp phát bộ nhớ động

```
#include <stdlib.h>
#include <stdio.h>
...
int *intPtr;
...
intptr = (int *) calloc (sizeof (int), 1000);
if ( intPtr == NULL )
{
  fprintf (stderr, "calloc failed\n");
  exit (EXIT_FAILURE);
}
```

Con trỏ và cấp phát bộ nhớ động

- Giải phóng bộ nhớ sau khi đã sử dụng xong: hàm free

  free(pointer);

  Trong đó pointer là con trỏ chứa địa chỉ đầu của vùng nhớ đã cấp phát

```
int *intPtr;
...
intptr = (int *) calloc (sizeof (int), 1000);
....
free(intptr);
```

Con trỏ và cấp phát bộ nhớ động

```
struct entry
{
    int value;
    struct entry *next;
};

struct entry *addEntry (struct entry *listPtr)
{
    // find the end of the list
    while ( listPtr->next != NULL )
        listPtr = listPtr->next;
    // get storage for new entry
    listPtr->next = (struct entry *) malloc (sizeof (struct entry));
    // add null to the new end of the list
    if ( listPtr->next != NULL )
        (listPtr->next)->next = (struct entry *) NULL;
    return listPtr->next;
}
```