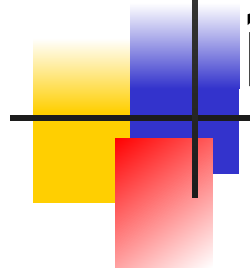




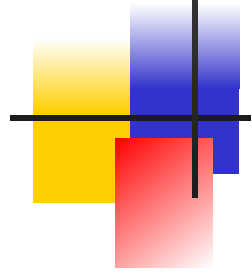
Con tr3

Bài 8



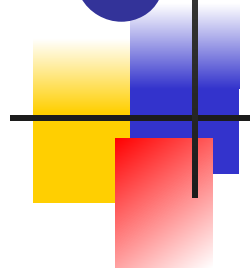
Mục tiêu bài học

- Tìm hiểu về con trỏ và khi nào thì sử dụng con trỏ
- Cách sử dụng biến con trỏ và các toán tử con trỏ
- Gán giá trị cho con trỏ
- Phép toán trên con trỏ
- So sánh con trỏ
- Con trỏ và mảng một chiều
- Con trỏ và mảng nhiều chiều
- Tìm hiểu cách cấp phát bộ nhớ



Con trỏ là gì?

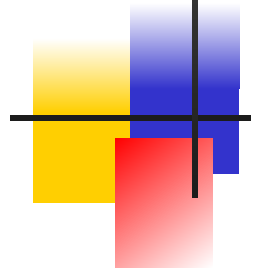
- Con trỏ là một biến, nó chứa địa chỉ ô nhớ của một biến khác
- Nếu một biến chứa địa chỉ của một biến khác, thì biến này được gọi là con trỏ *trở* đến biến thứ hai
- Con trỏ cung cấp phương thức truy xuất gián tiếp đến giá trị của một phần tử dữ liệu
- Các con trỏ có thể trỏ đến các biến có kiểu dữ liệu cơ bản như **int**, **char**, **double**, hay dữ liệu tập hợp như **mảng** hoặc **cấu trúc**.



Con trở được sử dụng để làm gì?

Các tình huống con trở có thể được sử dụng:

- Để trả về nhiều hơn một giá trị từ một hàm
- Để truyền mảng và chuỗi từ một hàm đến một hàm khác thuận tiện hơn
- Để làm việc với các phần tử của mảng thay vì truy xuất trực tiếp vào các phần tử này
- Để cập phát bộ nhớ và truy xuất bộ nhớ (Cập phát bộ nhớ trực tiếp)



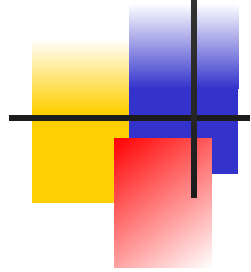
Biến con trỏ

- Khai báo con trỏ: chỉ ra một kiểu cơ sở và một tên biến được đặt trước bởi dấu *

Cú pháp khai báo tổng quát:

```
type  
*name;  
int  
*var2;
```

Ví dụ:



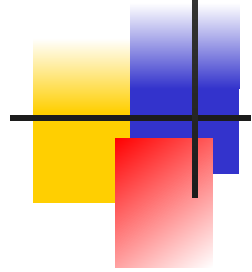
Các toán tử con trỏ

- Hai toán tử đặc biệt được sử dụng với con trỏ:
& và *****
- **&** là toán tử một ngôi và nó trả về địa chỉ ô nhớ của toán hạng
var2 =
- Toán tử ***** là toán tử hai ngôi, xung của toán tử **&**. Đây là toán tử một ngôi và nó trả về giá trị chứa trong vùng nhớ được trỏ đến bởi biến con trỏ
temp = *var2;



Gán trị đối với con trỏ

- Các giá trị có thể được gán cho con trỏ thông qua toán tử **&**.
ptr_var = &var;
- Ở đây địa chỉ của var được lưu vào biến ptr_var.
- Cũng có thể gán giá trị cho con trỏ thông qua một biến con trỏ khác trỏ có cùng kiểu.
ptr_var = &var;
ptr_var2 = ptr_var;

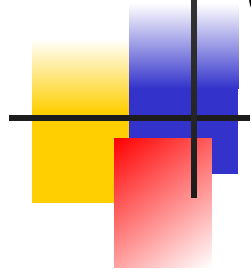


Gán trị đổi với con trỏ (tt)

- Có thể gán giá trị cho các biến thông qua con trỏ

```
*ptr_var = 10;
```

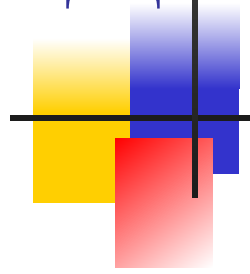
- Câu lệnh trên gán giá trị 10 cho biến var nếu ptr_var đang trỏ đến var



Phép toán con trỏ

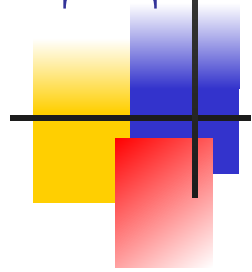
- Chỉ có thể thực hiện phép toán cộng và trừ trên con trỏ

```
int var, *ptr_var;
ptr_var = &var;
var = 500;
ptr_var++;
```
- Giả sử biến **var** được lưu trữ tại địa chỉ **1000**
- ptr_var** lưu giá trị 1000. Vì số nguyên có kích thước là 2 bytes, nên sau biểu thức “**ptr_var++;**” **ptr_var** sẽ có giá trị là 1002 mà không là 1001



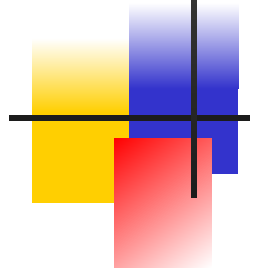
Phép toán con trỏ (tt)

<code>++ptr_var</code> or <code>ptr_var++</code>	Trỏ đến số nguyên kế tiếp đứng sau <code>var</code>
<code>--ptr_var</code> or <code>ptr_var--</code>	Trỏ đến số nguyên đứng trước <code>var</code>
<code>ptr_var + i</code>	Trỏ đến số nguyên thứ <code>i</code> sau <code>var</code>
<code>ptr_var - i</code>	Trỏ đến số nguyên thứ <code>i</code> trước <code>var</code>
<code>++*ptr_var</code> or <code>(*ptr_var)++</code>	Sẽ tăng trị var bởi 1
<code>*ptr_var++</code>	Sẽ tác động đến giá trị của số nguyên kế tiếp sau <code>var</code>



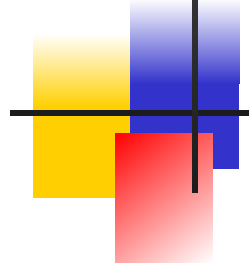
Phép toán con trỏ (tt)

- Mỗi lần con trỏ được tăng trị, nó trở đến ô nhớ của phần tử kế tiếp
- Mỗi lần con trỏ được giảm trị, nó trở đến ô nhớ của phần tử đứng trước nó
- Tất cả con trỏ sẽ tăng hoặc giảm trị theo kích thước của kiểu dữ liệu mà chúng đang trỏ đến



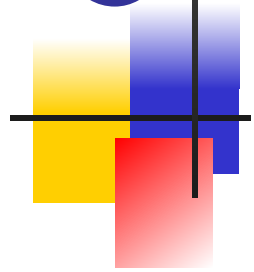
So sánh con trỏ

- Hai con trỏ có thể được so sánh trong một biểu thức quan hệ nếu chúng trỏ đến các biến có cùng kiểu dữ liệu
- Giả sử ptr_a và ptr_b là hai biến con trỏ trỏ đến các phần tử dữ liệu a và b. Trong trường hợp này, các phép so sánh sau là có thể:



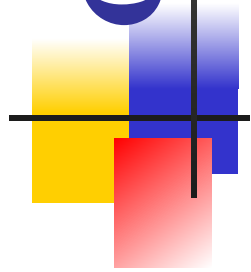
So sánh con trỏ (tt)

<code>ptr_a < ptr_b</code>	Trả về giá trị true nếu a được lưu trữ ở vị trí trước b
<code>ptr_a > ptr_b</code>	Trả về giá trị true nếu a được lưu trữ ở vị trí sau b
<code>ptr_a <= ptr_b</code>	Trả về giá trị true nếu a được lưu trữ ở vị trí trước b hoặc ptr_a và ptr_b trở đến cùng một vị trí
<code>ptr_a >= ptr_b</code>	Trả về giá trị true nếu a được lưu trữ ở vị trí sau b hoặc ptr_a và ptr_b trở đến cùng một vị trí
<code>ptr_a == ptr_b</code>	Trả về giá trị true nếu cả hai con trỏ ptr_a và ptr_b trở đến cùng một phân tử dữ liệu.
<code>ptr_a != ptr_b</code>	Trả về giá trị true nếu cả hai con trỏ ptr_a và ptr_b trở đến các phân tử dữ liệu khác nhau nhưng có cùng kiểu dữ liệu.
<code>ptr_a == NULL</code>	Trả về giá trị true nếu ptr_a được gán giá trị NULL (0)



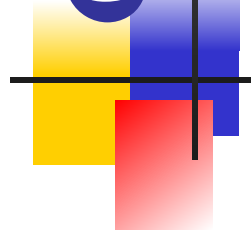
Con trỏ và mảng một chiều

- Địa chỉ của một phần tử mảng có thể được biểu diễn theo hai cách:
- Sử dụng ký hiệu & trước một phần tử mảng.
- Sử dụng một biểu thức trong đó chỉ số của phần tử được cộng vào tên của mảng.



Con tr  và m ng một chiều-ví dụ

```
#include<stdio.h>
void main() {
    static int ary[10]
    = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    for (i= 0; i<10; i++) {
        printf("\ni=%d, ary[i]=%d, *(ary+i)=%d" ,
            i, ary[i], *(ary + i));
        printf("&ary[i]=%X, ary+i=%X" , &ary[i],
            ary+i);
        /*%X gives unsigned hexadecimal*/
    }
}
```



Con trỏ và mảng một chiều-ví dụ tt

i=0	ary[i]=1	*(ary+i)=1	&ary[i]=194	ary+i = 194
i=1	ary[i]=2	*(ary+i)=2	&ary[i]=196	ary+i = 196
i=2	ary[i]=3	*(ary+i)=3	&ary[i]=198	ary+i = 198
i=3	ary[i]=4	*(ary+i)=4	&ary[i]=19A	ary+i = 19A
i=4	ary[i]=5	*(ary+i)=5	&ary[i]=19C	ary+i = 19C
i=5	ary[i]=6	*(ary+i)=6	&ary[i]=19E	ary+i = 19E
i=6	ary[i]=7	*(ary+i)=7	&ary[i]=1A0	ary+i = 1A0
i=7	ary[i]=8	*(ary+i)=8	&ary[i]=1A2	ary+i = 1A2
i=8	ary[i]=9	*(ary+i)=9	&ary[i]=1A4	ary+i = 1A4
i=9	ary[i]=10	*(ary+i)=10	&ary[i]=1A6	ary+i = 1A6



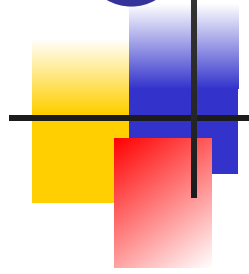
Con trỏ và mảng đa chiều

- Mảng hai chiều có thể được định nghĩa như là một con trỏ tới một nhóm các mảng một chiều liên tiếp nhau
- Khai báo một mảng hai chiều có thể như sau:

```
data_type (*ptr_var) [expr 2];
```

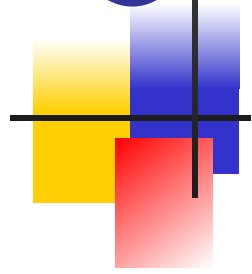
thay vì

```
data_type (*ptr_var) [expr 1] [expr 2];
```



Con trỏ và chuỗi

```
#include <stdio.h>
#include <string.h>
void main () {
    char a, str[81], *ptr;
    printf("\nEnter a sentence:");
    gets(str);
    printf("\nEnter character to search for:");
    a = getche();
    ptr = strchr(str,a);
    /* return pointer to char*/
    printf( "\nString starts at address: %u",str);
    printf("\nFirst occurrence of the character is at
address: %u ",ptr);
    printf("\n Position of first occurrence(starting
from 0) is: % d", ptr_str);
}
```



Con trỏ và chuỗi (tt)

```
Enter a sentence: We all live in a yellow submarine
Enter character to search for: Y
String starts at address: 65420.
First occurrence of the character is at address: 65437.
Position of first occurrence (starting from 0) is: 17
```



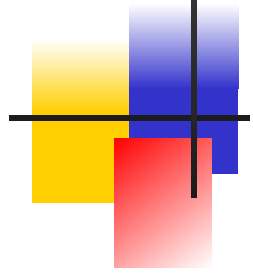
Cấp phát bộ nhớ

Hàm `malloc()` là một trong các hàm được sử dụng thường xuyên nhất để thực hiện việc cấp phát bộ nhớ từ vùng nhớ còn tự do.

Tham số của hàm `malloc()` là một số nguyên xác định số bytes cần cấp phát.

Cấp phát bộ nhớ (tt)

```
#include<stdio.h>
#include<malloc.h>
void main()
{
    int *p,n,i,j,temp;
    printf("\n Enter number of elements in the array :");
    scanf("%d",&n);
    p=(int*)malloc(n*sizeof(int));
    for(i=0;i<n;++i) {
        printf("\nEnter element no. %d:",i+1);
        scanf("%d",p+i);
    }
    for(i=0;i<n-1;++i)
        for(j=i+1;j<n;++j) {
            if(*p+i)>*(p+j)) {
                temp=*(p+i);
                *(p+i)=*(p+j);
                *(p+j)=temp;
            }
        }
    for(i=0;i<n;++i)
        printf("%d\n",*(p+i));
}
```



Hàm free()

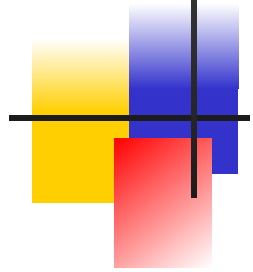
Hàm `free()` được sử dụng để giải phóng bộ nhớ khi nó không cần dùng nữa.

Cú pháp:

```
void free(void*ptr);
```

Hàm này giải phóng không gian được trả bởi *ptr*, để dùng cho tương lai.

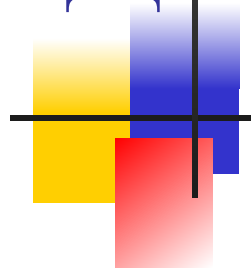
ptr phải được dùng trước đó với lời gọi hàm `malloc()`, `calloc()`, hoặc `realloc()`.



Hàm free() - tt

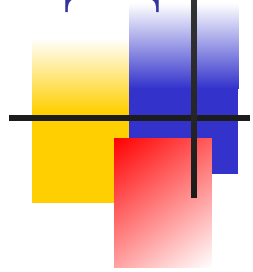
```
#include <stdio.h>
#include <stdlib.h>
/*required for the malloc and free functions*/
int main(){
    int number;
    int *ptr;
    int i;
    printf("How many ints would you like store? ");
    scanf("%d", &number);
    ptr = (int *) malloc (number*sizeof(int));
    /*allocate memory */
    if(ptr!=NULL) {
        for(i=0 ; i<number ; i++){
            *(ptr+i) = i;
        }
    }
```

Còn tiếp...



Hàm free() - tt

```
for (i=number ; i>0 ; i--) {  
    printf("%d\n", *(ptr+(i-1)));  
    /* print out in reverse order */  
}  
free(ptr); /* free allocated memory */  
return 0;  
  
} else {  
    printf("\nMemory allocation failed -  
not enough memory.\n");  
    return 1;  
}  
}
```

Hàm calloc()

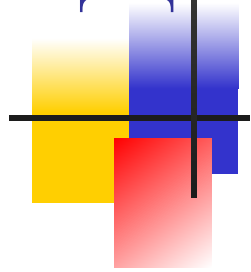
calloc tương tự như **malloc**, nhưng điểm khác biệt chính là mặc nhiên giá trị 0 được lưu vào không gian bộ nhớ vừa cấp phát

calloc yêu cầu hai tham số

- Tham số thứ nhất là số lượng các biến cần cấp phát bộ nhớ
- Tham số thứ hai là kích thước của mỗi biến

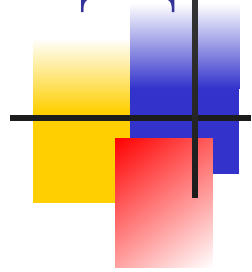
Cú pháp:

```
void *calloc( size_t num, size_t size );
```



Hàm calloc() - tt

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    float *calloc1, *calloc2;
    int i;
    calloc1 = (float *) calloc(3, sizeof(float));
    calloc2 = (float *) calloc(3, sizeof(float));
    if(calloc1!=NULL && calloc2!=NULL) {
        for(i=0 ; i<3 ; i++){
            printf("calloc1[%d] holds %05.5f ", i,
                calloc1[i]);
            printf("\ncalloc2[%d] holds %05.5f",
                i, *(calloc2+i));
        }
    }
    tiếp.....
    Còn
```



Hàm calloc() - tt

```
free(calloc1);  
free(calloc2);  
return 0;  
}  
else{  
    printf("Not enough memory\n");  
    return 1;  
}  
}
```



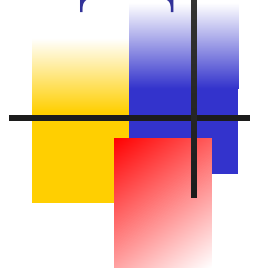
Hàm realloc()

Có thể cấp phát lại cho một vùng đã được cấp (thêm/bớt số bytes) bằng cách sử dụng hàm **realloc**, mà không làm mất dữ liệu.

realloc nhận hai tham số

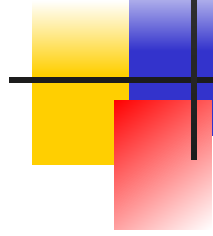
- Tham số thứ nhất là con trỏ tham chiếu đến bộ nhớ
- Tham số thứ hai là tổng số byte muốn cấp phát
- Cú pháp:

```
void *realloc( void *ptr, size_t size );
```



Hàm realloc() - tt

```
#include<stdio.h>
#include <stdlib.h>
int main() {
    int *ptr;
    int i;
    ptr = (int *)calloc(5, sizeof(int *));
    if(ptr!=NULL) {
        *ptr = 1; *(ptr+1) = 2;
        ptr[2] = 4; ptr[3] = 8; ptr[4] = 16;
        ptr = (int *)realloc(ptr, 7*sizeof(int));
        if(ptr!=NULL) {
            printf("Now allocating more memory...\n");
            ptr[5] = 32; /* now it's legal! */
            ptr[6] = 64;
```



Hàm realloc() - tt

```
for(i=0;i<7;i++) {  
    printf("ptr[%d] holds %d\n", i, ptr[i]);  
}  
    realloc(ptr,0);  
/* same as free(ptr); - just fancier! */  
    return 0;  
}  
else {  
    printf("Not enough memory-realloc failed.\n");  
    return 1;  
}  
}  
else {  
    printf("Not enough memory-calloc failed.\n");  
    return 1;  
}  
}
```