Pointer & Structure

Course Code: CSC 2106 Course Title: Data Structure (Theory)

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**2.2**	**Week No:**	**2**	**Semester:**	**Fall 24**
Lecturer:	*Mashiour Rahman (mashiour@aiub.edu)*				

Lecture Outline

1. Pointer
   i. Variable
   ii. Example
   iii. Pointer & Array
   iv. Void Pointer
   v. Null Pointer
   vi. Dynamic Memory Allocation
   vii. Pointer & Function
   viii. Pointer, Array & Function
   ix. Pointers & Initialization
2. Structure
   i. Definition
   ii. Defining Structure in C++
   iii. Declaring Variable of Structure
   iv. Access Structure Member
   v. Initializing Structure Variable
   vi. Some Facts about Structure
   vii. Nested Structure
   viii. Self-referential Structure
3. Books
4. References

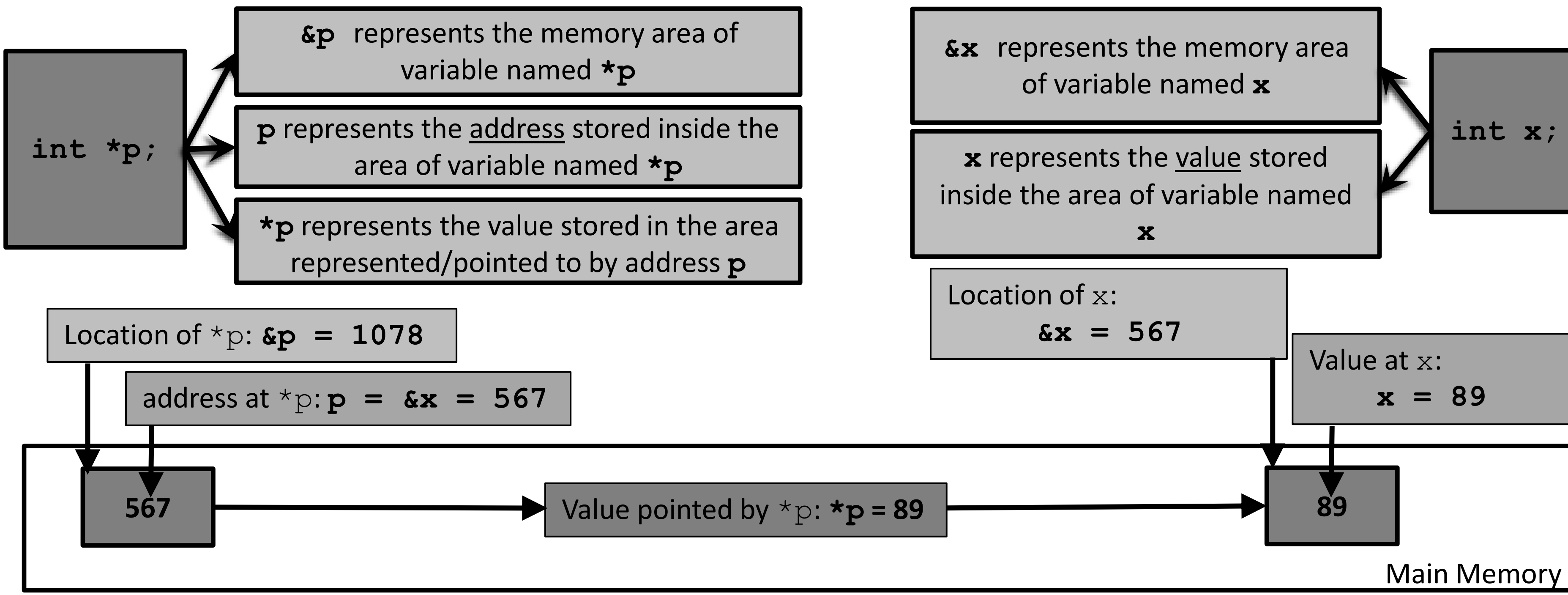
Pointer

Variable

- The computer access its own memory not by using variable names but by using a memory map where each location of memory is uniquely defined by a number, called the *address*. Pointers are a very powerful, but primitive facility to avail that address. To understand pointer let us go through the concept of variables once more.
- A *variable* is an area of *memory* that has been given a name. For example: `int x;` is an area of memory that has been given the name `x`. The instruction `x=10;` stores the data value `10` in the area of memory named `x`. The instruction `&x` returns the *address* of the location of variable `x`.
- A *pointer* is a variable that *stores the location of a memory/variable*. A pointer has to be declared. For example: `int *p;` Adding an asterisk (called the *de-referencing* operator) in front of a variable's name declares it to be a pointer to the declared type.
- `int *p;` is a pointer – which can store an address of a memory location where an integer value can be stored or which can store an address of the memory location of an integer variable.
- For example: `int *p , q;` declares `p`, a pointer to `int`, and `q` an `int` and the instruction: `p=&q;` stores the address of `q` in `p`. After this instruction, conceptually, `p` is *pointing* at `q`.

Variable

- After declaring a pointer `*p` variable, it can be used like any other variable. That is, `p` stores the *address* or *pointer*, to another variable; `&p` gives the address of the pointer variable itself; and `*p` is the *value* stored in the variable that `p` *points* at.



Example

```
// Understanding pointer variable
void main( void )
{
 int x = 10;
 int *p = &x;
 int y = *p;
 cout <<"Address of integer variable x: "<< &x <<"\n";
 cout <<"Value stored in the memory area of x: "<< x <<"\n";
 cout <<"Address of integer pointer variable *p: "<< &p <<"\n";
 cout <<"Address stored in the area of pointer *p: "<< p<<"\n";
 cout <<"Address of integer variable y: "<< &y <<"\n";
 cout <<"Value pointed to by the pointer *p: "<< *p <<"\n";
 cout <<"Value stored in the memory area of variable y: "<< y <<"\n";
}
```

```
Address of integer variable x: 0x8fbbfff0
Value stored in the memory area of x: 10
Address of integer pointer variable *p: 0x8fbbfff4
Address stored in the area of pointer *p: 0x8fbbfff0
Address of integer variable y: 0x8fbbfff8
Value pointed to by the pointer *p: 10
Value stored in the memory area of variable y: 10
```

variable	Memory Address	value	
int x	0x8f86fff0	10	x=10 &x=0x8f86fff0
	0x8f86fff1		
	0x8f86fff2		
	0x8f86fff3		
int *p	0x8f86fff4	0x8f86fff0	p=0x8f86fff0 &p=0x8f86fff4 *p=*(0x8f86fff0) =*(&x)=x=10
	0x8f86fff5		
	0x8f86fff6		
	0x8f86fff7		
int y	0x8f86fff8	10	y=*p=10 &y=0x8f86fff8
	0x8f86fff9		
	0x8f86fffa		
	0x8f86fffb		

Pointer & Array

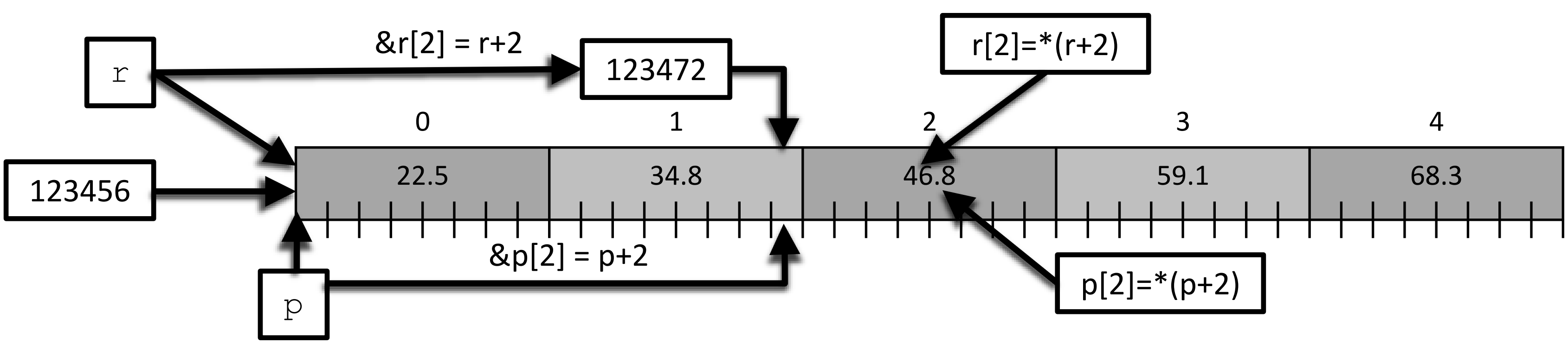
An array is simply a block of memory. An array can be accessed with pointers as well as with `[]` square brackets. *The name of an array variable is a pointer to the first element in the array.* So, any operation that can be achieved by array subscripting can also be done with pointers or vice-versa.

```
void main( void )
{
 float r[5] = {22.5,34.8,46.8,59.1,68.3};
 cout <<"1st element: "<< r[0] <<"\n";
 cout <<"1st element: "<< *r <<"\n";
 cout <<"3rd element: "<< r[2] <<"\n";
 cout <<"3rd element: "<< *(r+2)<<"\n";
 float *p;
 p = r; //&r[0]
 cout <<"1st element: "<< p[0] <<"\n";
 cout <<"1st element: "<< *p <<"\n";
 cout <<"3rd element: "<< p[2]<<"\n";
 cout <<"3rd element: "<< *(p+2)<<"\n";
 for(int i=0; i<5; i++, p++)
  cout <<"Element "<<(i+1)<<" is: "<<*p<<"\n";
}
```

```
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
1st element: 22.5
1st element: 22.5
3rd element: 46.8
3rd element: 46.8
Element 1 is: 22.5
Element 2 is: 34.8
Element 3 is: 46.8
Element 4 is: 59.1
Element 5 is: 68.3
```

Pointer & Array

- The array `float r[5];` or the pointer variable `*p` (after `p=r`) is a pointer to the first floating point number in the declared array.
- The 1st element of the array, `22.3`, can be accessed by using: `r[0]`, `p[0]`, `*r` or `*p`.
- The 3rd element, `46.8`, could be accessed by using: `r[2]`, `p[2]`,`*(r+2)` or `*(p+2)`.
- Now, let's examine the notation `(r+2)` and `(p+2)`.



- Assuming the starting address of the array numbers is 123456 –

`r[0]=(r+0)` starts at address, `r+0*sizeof(float)` = 123456 + 0 * 8 = 123456,
`r[1]=(r+1)` starts at address, `r+1*sizeof(float)` = 123456 + 1 * 8 = 123464,
`r[2]=(r+2)` starts at address, `r+2*sizeof(float)` = 123456 + 2 * 8 = 123472.

Void Pointer

```
// increaser
void increase(void *data, int psize){
  if ( psize == sizeof(char) ){
     char *pchar;
     pchar=(char*)data;
     ++(*pchar);
  }
  else if (psize == sizeof(int)){
     int *pint;
     pint=(int*)data;
     ++(*pint);
  }
}
void main (void){
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
}
```

```
Y, 1603
```

- The `void` type of pointer is a special type of pointer which represents the absence of type. So `void` pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).
- This allows `void` pointers to point to any data type, `int`, `float`, `char`, `double` or any type of array.
- But the data pointed by them cannot be directly dereferenced, since we have no type to dereference to.
- So need to *cast* the address in the `void` pointer to some other pointer type that points to a concrete data type before dereferencing it.

Void Pointer

```
1  // increaser
2  void increase(void *data, int psize){
3    if ( psize == sizeof(char) ){
4       char *pchar;
5       pchar=(char*)data;
6       ++(*pchar);
7    }
8    else if (psize == sizeof(int)){
9       int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

- We start with the `main` where two variables `char a` and `int b` is created with the values `'x'` and `1602` respectively (line 15-16).

- Line 17 calls the function `increase` with address of `a` and the size of `a` as parameters.

- `sizeof(a)=sizeof(char) = 1`, as `char` type is one byte long.

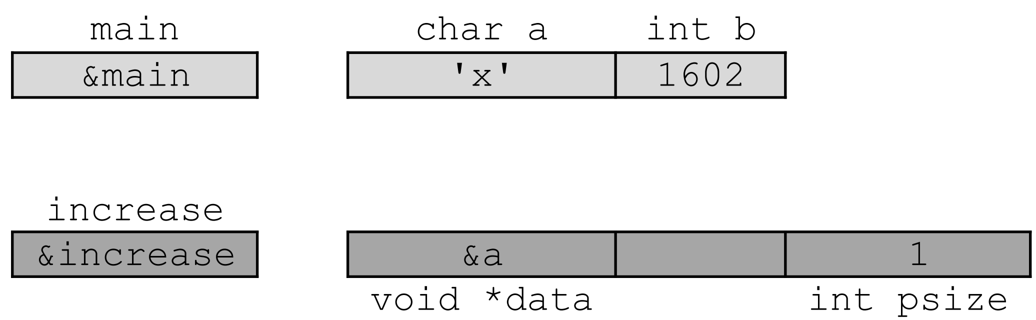
main	char a	int b
&main	'x'	1602

Void Pointer

```
1  // increaser
2  void increase(void *data, int psize){
3    if ( psize == sizeof(char) ){
4       char *pchar;
5       pchar=(char*)data;
6       ++(*pchar);
7    }
8    else if (psize == sizeof(int)){
9       int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

- Line 17 gives the control to the function `increase` and it creates two (parameter) variables `void *data` and `int psize` assigned with the address value of `a` and `sizeof(a)=1` of `main` respectively.

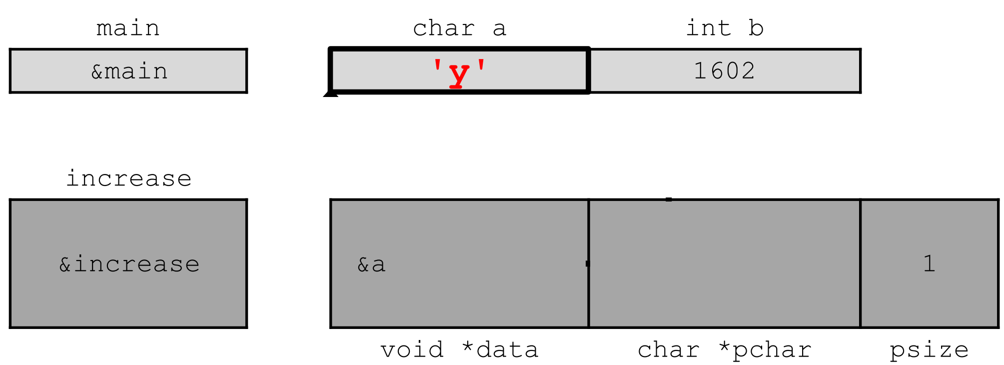
- Though `*data` contains the address of `a` in `main`, this address cannot be accessed using `*data` with type mismatch.



Void Pointer

```
1  // increaser
2  void increase(void *data, int psize){
3    if ( psize == sizeof(char) ){
4       char *pchar;
5       pchar=(char*)data;
6       ++(*pchar);
7    }
8    else if (psize == sizeof(int)){
9       int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

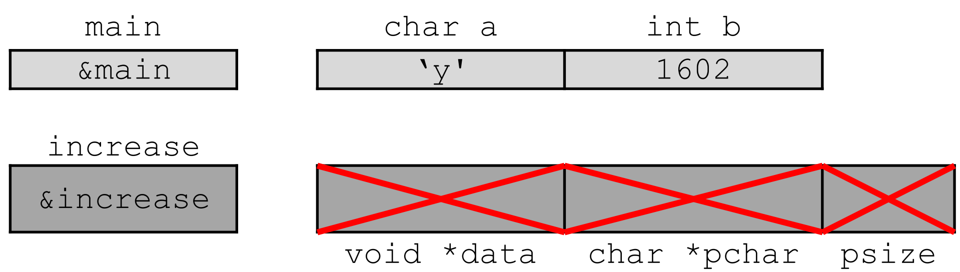
- ❑ With the `true` value of the conditional statement `psize==sizeof(char)` another new pointer variable `char *pchar` is created and is assigned the value `*data` (line 3-5)
- ❑ As `*data` has no type, it must be ***type casted*** to `(char*)` before being assigned (line 5).
- ❑ With the statement `++(*pchar);` pointer variable `*pchar`, pointing to `a` of `main`, is increased by one. So, the value of a in `main` is changed to `'y'` (the `ASCII` value is increased by one) (line 6).



Void Pointer

```
1  // increaser
2  void increase(void *data, int psize){
3    if ( psize == sizeof(char) ){
4       char *pchar;
5       pchar=(char*)data;
6       ++(*pchar);
7    }
8    else if (psize == sizeof(int)){
9       int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

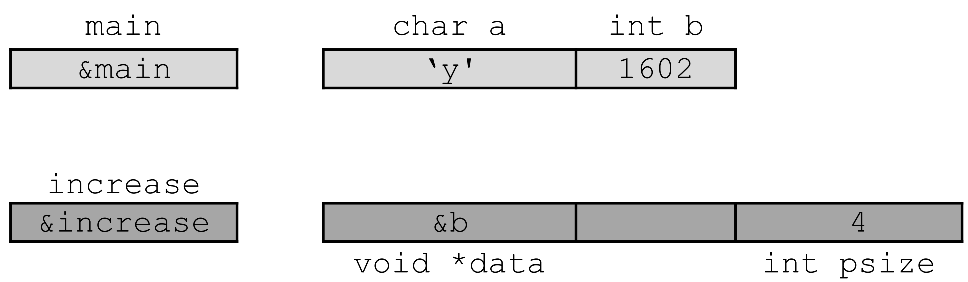
- Before exiting the function `increase`, the variables created by increase is destroyed.
- Then the control goes back to the function main (in line 17).
- So, we see the value `a` in `main` is changed to `'y'`.



Void Pointer

```
1  // increaser
2  void increase(void *data, int psize){
3    if ( psize == sizeof(char) ){
4       char *pchar;
5       pchar=(char*)data;
6       ++(*pchar);
7    }
8    else if (psize == sizeof(int)){
9       int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

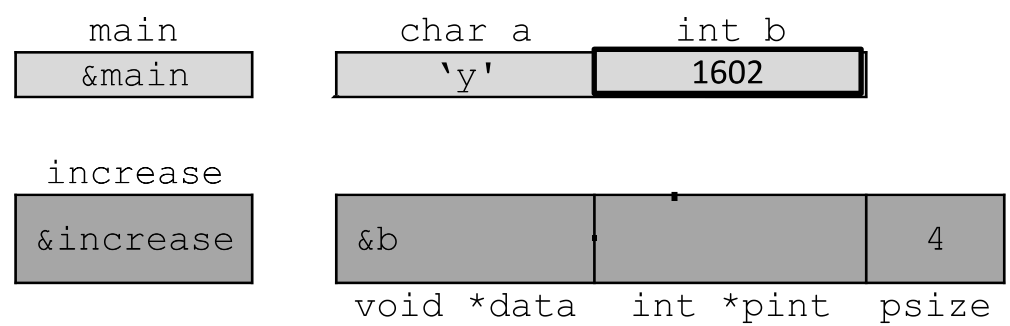
- Line 18 calls the function `increase` with address of `b` and the size of `b` as parameters.
- `sizeof(b)= sizeof(int)= 4`, as `int` type is four bytes long.
- Now the control goes to the function `increase` and it creates two (parameter) variables `void *data` and `int psize` assigned with the address value of `a` and `sizeof(b)=4` of `main` respectively.



Void Pointer

```
 1 // increaser
 2 void increase(void *data, int psize){
 3   if ( psize == sizeof(char) ){
 4      char *pchar;
 5      pchar=(char*)data;
 6      ++(*pchar);
 7   }
 8   else if (psize == sizeof(int)){
 9      int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

- With the `true` value of the conditional statement `psize==sizeof(int)`, a new pointer variable `int *pint` is created and assigned to the value, `*data` (line 8-10) .
- Though `*data` contains the address of `b` in `main`, this address cannot be accessed using `*data` with type mismatch (line 10).
- As `*data` has no type, it must be type casted to `int *` before being assigned (line 10).

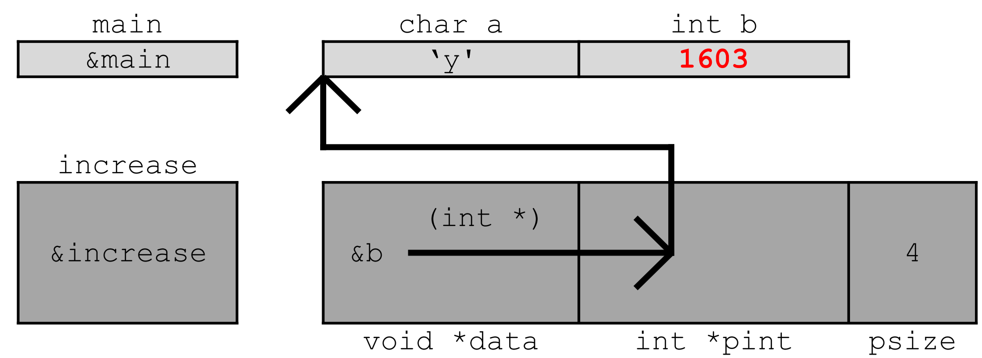


Void Pointer

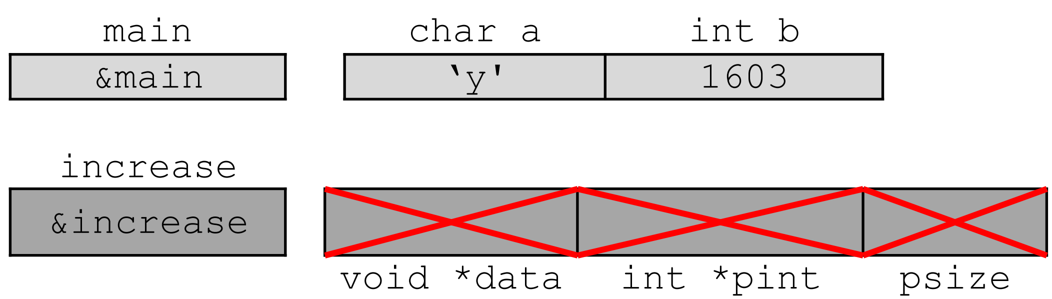
```
 1 // increaser
 2 void increase(void *data, int psize){
 3   if ( psize == sizeof(char) ){
 4      char *pchar;
 5      pchar=(char*)data;
 6      ++(*pchar);
 7   }
 8   else if (psize == sizeof(int)){
 9      int *pint;
10      pint=(int*)data;
11      ++(*pint);
12   }
13 }
14 void main (void){
15   char a = 'x';
16   int b = 1602;
17   increase (&a,sizeof(a));
18   increase (&b,sizeof(b));
19   cout << a << ", " << b << endl;
20 }
```

Y, 1603

- With the statement `++(*pint);` pointer variable `*pint`, pointing to a of `main`, is increased by one. So, the value of a in `main` is changed to `1603` (line 11).



- Before exiting the function increase, the variables created by increase is destroyed. Then the control goes back to the function main (in line 17). So, we see the value `b` is changed to 1063.



Null Pointer

- A `NULL` pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type.

```
int * p;
p = 0;  //can also write, p = NULL;
/* p has a null pointer value */
```

- Do not confuse `null` pointers with `void` pointers. A `null` pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a `void` pointer is a special type of pointer that can point to somewhere without a specific type.

- One refers to the value stored in the pointer itself and the other to the type of data it points to.

Dynamic Memory Allocation

- The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into an executable form. The size of array declared initially can be sometimes insufficient and sometimes more than required. Also, what if we need a variable amount of memory that can only be determined during runtime?

- Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

- C++ integrates the operators `new` and `delete` for *dynamic memory* allocation.

Dynamic Memory Allocation: Operators `new` And `new[]`

- In order to request dynamic memory we use the operator `new`. `new` is followed by a data type specifier. If a sequence of more than one memory block is required, the data type specifier is followed by the number of these memory blocks within brackets `[]`. It returns a pointer to the beginning of the new block of memory allocated. Syntax:

```
pointer = new vtype;
pointer = new vtype [number_of_elements];
```

- The first expression is used to allocate memory to contain one single element of type `vtype`. The second one is used to assign a block (an array) of elements of type `vtype`, where `number_of_elements` is an integer value representing the amount of these.

Dynamic Memory Allocation: Operators `delete` And `delete[]`

- Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator `delete`, whose format is:

```
delete pointer;
delete [] pointer;
```

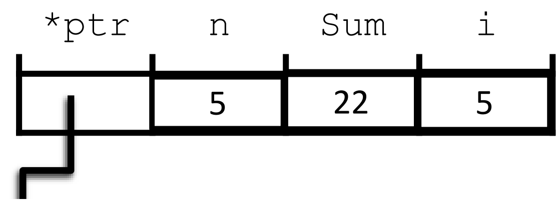
- The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

- The value passed as argument to delete must be either a pointer to a memory block previously allocated with `new`, or a `null` pointer (in the case of a `null` pointer, `delete` produces no effect).

Dynamic Memory Allocation: Example

```
// new, delete
void main(void){
    int n, i,*ptr, sum=0;
    cout << "# of elements: ";
    cin >> n;           //input 5
    ptr = new (nothrow) int[n];
    if(ptr==NULL){   //ptr==0
      cout << "Error! not
allocated.";
      return 1;
    }
    cout << "Enter elements:\n";
    for(i=0;i<n;++i)
    {   //input 2 6 7 4 3
        cin >> *(ptr+i);   //ptr[i]
        sum += *(ptr+i);
    }
    cout << "Sum = " << sum;
    delete [] (ptr);
    //memory de-allocated
}
```

```
# of elements: 5
Enter elements:
2
6
7
4
3
Sum = 22
```



Pointer & Function

```
1  /* Swap two numbers using function. */
2  void swap(int *a, int *b);
3  void main(void){
4    int num1=5,num2=10;
5    swap(&num1,&num2);
6    /* address of num1, num2 is passed */
7    cout<<"Number1 = "<<num1<<"\n";
8    cout<<"Number2 = "<<num2;
9  }
10 void swap(int *a, int *b){
11 //a,b points to &num1,&num2 respectively
12   int t;
13   t = *a;
14   *a = *b;
15   *b = t;
16 }
```

```
Number1 = 10
Number2 = 5
```

- Passing arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function.
- Pointer arguments enable a function to access and change objects in the function that called it. Let's consider the example on the left.
- The program starts with function main getting the control and creating two variables `num1` and `num2` with values `5` and `10` respectively (line 4).

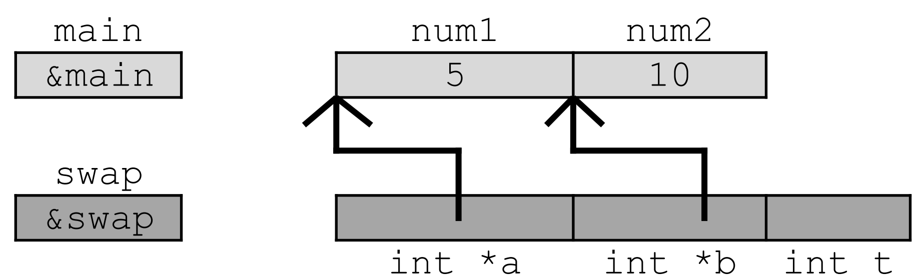
main	int num1	int num2
&main	5	10

Pointer & Function

```
1 /* Swap two numbers using function. */
2 void swap(int *a, int *b);
3 void main(void){
4   int num1=5,num2=10;
5   swap(&num1,&num2);
6   /* address of num1, num2 is passed */
7   cout<<"Number1 = "<<num1<<"\n";
8   cout<<"Number2 = "<<num2;
9 }
10 void swap(int *a, int *b){
11 //a,b points to &num1,&num2 respectively
12   int t;
13   t = *a;
14   *a = *b;
15   *b = t;
16 }
```

```
Number1 = 10
Number2 = 5
```

- Function `main` calls the function `swap` with two parameter values `&num1` and `&num2` (line 5).
- So the <u>address</u> of `num1` and `num2` is send through the parameter of `swap`.
- Now the control goes to the function `swap` and it creates two pointer (parameter) variables `*a` and `*b` assigned with the address values of `num1` and `num2` of `main` respectively.
- Another variable `t` is created (line 12).

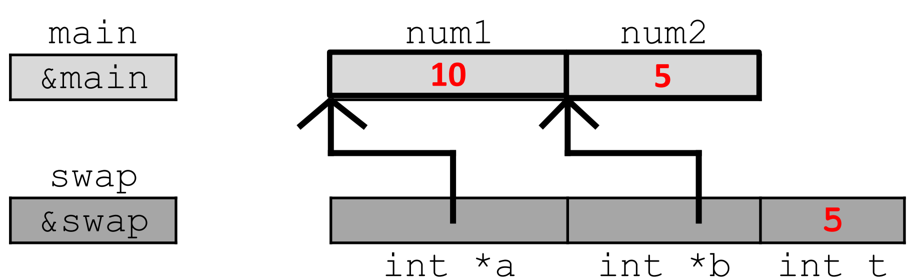


Pointer & Function

```
1 /* Swap two numbers using function. */
2 void swap(int *a, int *b);
3 void main(void){
4   int num1=5,num2=10;
5   swap(&num1,&num2);
6   /* address of num1, num2 is passed */
7   cout<<"Number1 = "<<num1<<"\n";
8   cout<<"Number2 = "<<num2;
9 }
10 void swap(int *a, int *b){
11 //a,b points to &num1,&num2 respectively
12   int t;
13   t = *a;
14   *a = *b;
15   *b = t;
16 }
```

```
Number1 = 10
Number2 = 5
```

- With the statement `t=*a;` variable `t` is assigned to the value, `num1` of `main`, pointed to by pointer `*a` (line 13).
- With the statement `*a = *b;` pointer variable `*a`, pointing to `num1` of `main`, is assigned to the value, `num2` in `main`, pointed to by pointer `*b` (line 14).
- With the statement `*b = t;` pointer variable `*b`, pointing to `num2` of `main`, is assigned to the value of `t` (line 15).

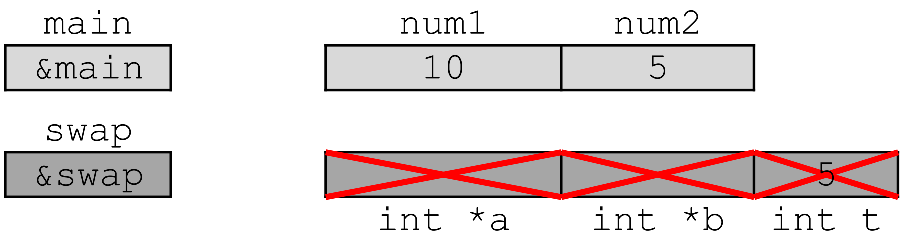


Pointer & Function

```
1  /* Swap two numbers using function. */
2  void swap(int *a, int *b);
3  void main(void){
4     int num1=5,num2=10;
5     swap(&num1,&num2);
6     /* address of num1, num2 is passed */
7     cout<<"Number1 = "<<num1<<"\n";
8     cout<<"Number2 = "<<num2;
9  }
10 void swap(int *a, int *b){
11 //a,b points to &num1,&num2 respectively
12    int t;
13    t = *a;
14    *a = *b;
15    *b = t;
16 }
```

```
Number1 = 10
Number2 = 5
```

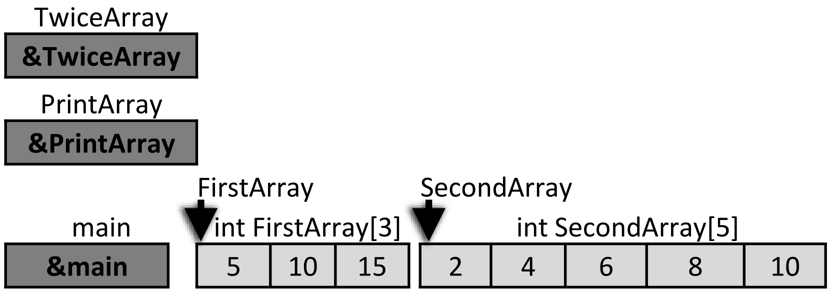
- Any changes we make to `*a` and `*b` in function `swap`, will change the values of `num1` and `num2` respectively in function `main` as `a` is pointing to `num1` and `b` is pointing to `num2`.
- Before exiting the function `swap`, the variables created by `swap` is destroyed. Then the control goes back to the function `main` (in line 5). But nothing changes for the values of the variables `num1` and `num2` of `main` due to the destruction of the variables `*a` and `*b`. Because, destruction only destroys the space provided for `*a` and `*b`. It does not destroy the space it was pointing to. Whatever changes were made in swap by `*a` and `*b` remains.



Pointer, Array & Function

```
1 // arrays as parameters
2 void TwiceArray (int arg[], int length) {
3   for (int n=0; n<length; n++) arg[n] *= 2;
4 }
5 void PrintArray(const int arg[], int
6 length){
7   for (int n=0; n<length; n++)
8     cout<<arg[n];
9   cout<<endl;
10 }
11 void main (void){
12   int FirstArray[3] = {5, 10, 15};
13   int SecondArray[] = {2, 4, 6, 8, 10};
14   TwiceArray (FirstArray,3);
15   PrintArray (FirstArray,3);
16   PrintArray (SecondArray,5);
   }
```

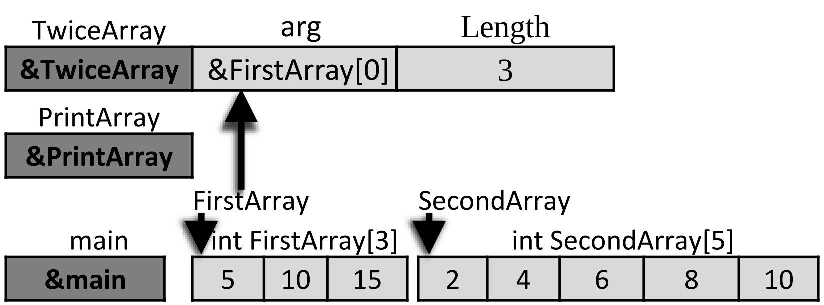
- Then we start with the function `main` where two arrays, `FirstArray` with 3 elements and `SecondArray` with 5 elements, are declared, created, and initialized (line 11-12).
- Both these identifiers will hold the starting address/location of their elements.
- `FirstArray` holds the location of the first element, `&FirstArray[0]` and
- `SecondArray` holds the location of the first element, `&SecondArray[0]`



Pointer, Array & Function

```
1 // arrays as parameters
2 void TwiceArray (int arg[], int length) {
3   for (int n=0; n<length; n++) arg[n] *= 2;
4 }
5 void PrintArray(const int arg[], int
6 length){
7   for (int n=0; n<length; n++)
8     cout<<arg[n];
9   cout<<endl;
10 }
11 void main (void){
12   int FirstArray[3] = {5, 10, 15};
13   int SecondArray[] = {2, 4, 6, 8, 10};
14   TwiceArray (FirstArray,3);
15   PrintArray (FirstArray,3);
16   PrintArray (SecondArray,5);
   }
```

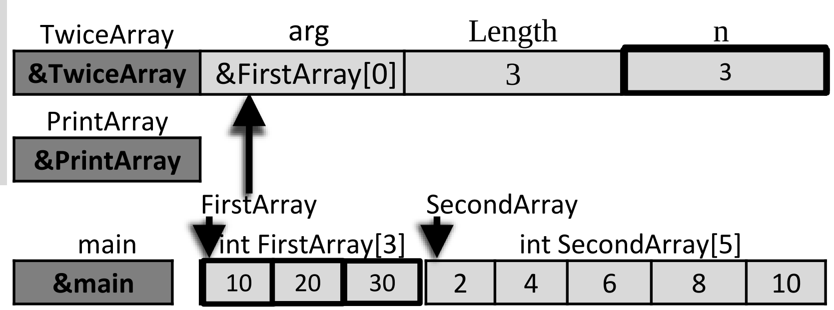
- Then, the `TwiceArray` is called with the parameters `FirstArray`, the starting location of the array itself or the location of the first element is passed to the parameter `int arg[]` and value `3` to the parameter `length` (line 13).
- The identifier `arg` holds the starting address of the `FirstArray` of the function main. As `arg` itself is an array, `arg` behaves like an array.
- The control is transferred from function main to function `TwiceArray` (line 2).



Pointer, Array & Function

```
1 // arrays as parameters
2 void TwiceArray (int arg[], int length) {
3   for (int n=0; n<length; n++) arg[n] *= 2;
4 }
5 void PrintArray(const int arg[], int
6 length){
7   for (int n=0; n<length; n++)
8     cout<<arg[n];
9   cout<<endl;
10 }
11 void main (void){
12   int FirstArray[3] = {5, 10, 15};
13   int SecondArray[] = {2, 4, 6, 8, 10};
14   TwiceArray (FirstArray,3);
15   PrintArray (FirstArray,3);
16   PrintArray (SecondArray,5);
  }
```

- Inside function `TwiceArray` another variable `n` is declared. Using `n` in `for` loop `3` elements of `arg` are made twice of their original (line 3).
- As `arg` represents the array `FirstArray` on `main`, the `3` elements of the `FirstArray` are actually made twice.
- Hypothetically,
- `FirstArray[n]`←`arg[n]*=2`.



Pointer, Array & Function

```
1 // arrays as parameters
2 void TwiceArray (int arg[], int length) {
3   for (int n=0; n<length; n++) arg[n] *= 2;
4 }
5 void PrintArray(const int arg[], int
6 length){
7   for (int n=0; n<length; n++)
8     printf("%d ", arg[n]);
9   printf("\n");
10 }
11 void main (void){
12   int FirstArray[3] = {5, 10, 15};
13   int SecondArray[] = {2, 4, 6, 8, 10};
14   TwiceArray (FirstArray,3);
15   PrintArray (FirstArray,3);
16   PrintArray (SecondArray,5);
   }
```

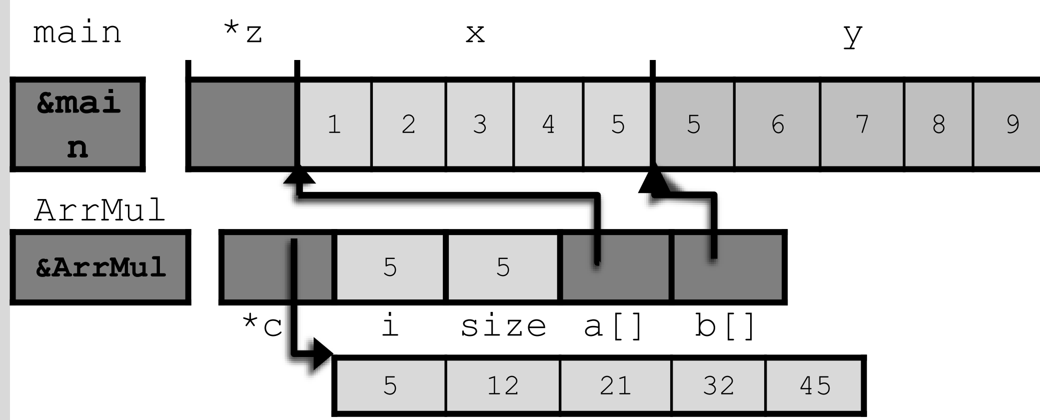
- Before exiting from the function `TwiceArray`, all the variables (`arg`, `length`, `n`) are destroyed and the control is returned to the `main` (line 13). Values of `FirstArray` changed in `TwiceArray` remains.
- Next the function `PrintArray` is called to print the elements of the `FirstArray` and `SecondArray` consecutively. Here, it works as same in terms of parameter passing. Except the parameter `arg` in `PrintArray` which is declared as `constant` variable.
- As we do not need to change any elements of the array inside `PrintArray`, the parameter `arg` is declared as `constant` variable.
- This is how any function (`main` in this example) can protect its data array (`FirstArray`) from being changed by another function (`PrintArray`).

Pointer, Array & Function

```
1  //Array Multiplication
2  int *ArrMul(int a[], int b[], int size){
3    int i,*c = new int[size]; //c[5]
4    for(i=0; i<size; i++) c[i] = a[i] * b[i];
5    return c;
6  }
7  void PrintArr(int *a, int size){
8    for(int i=0; i<size; i++)
9  cout<<a[i]<<"\t";
10   cout << "\n ";
11 }
12 void main(void){
13   int *z, x[5]={1,2,3,4,5},
14 y[5]={5,6,7,8,9};
15   z = ArrMul(x, y, 5);
16   cout <<"First array:\n";
17   PrintArr( x, 5 );
18   cout <<"second array:\n";
19   PrintArr( y, 5 );
20   cout <<"result array:\n";
21   PrintArr( z, 5 );
     delete [] (z); //memory deallocated. If
you look carefully, we are deallocating the
same memory that we allocated, because that
memory was passed to variable z from
variable c;
}
```

```
First array:
1    2    3    4    5
second array:
5    6    7    8    9
result array:
5    12   21   32   45
```

- Two array `x` and `y` with five elements are multiplied index wise using the function `ArrMul`.
- Function `ArrMul` (line 2-6) dynamically allocates memory for the resultant array `c` of the multiplication.

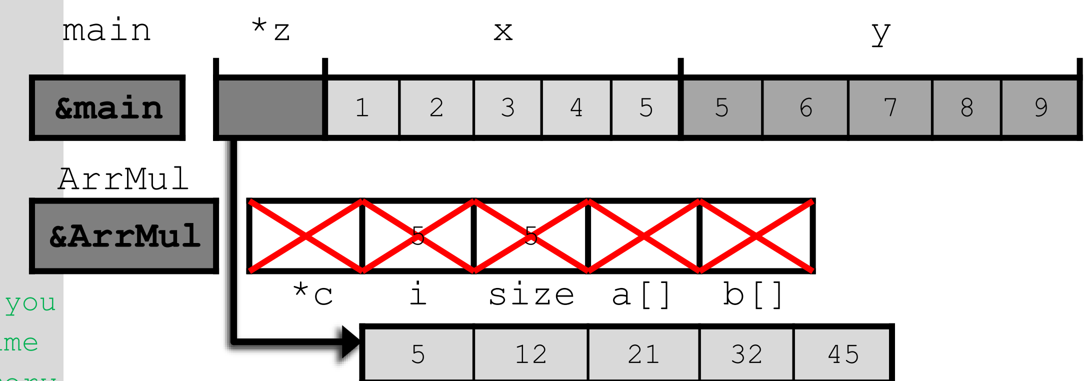


Pointer, Array & Function

```
1 //Array Multiplication
2 int *ArrMul(int a[], int b[], int size){
3   int i,*c = new int[size]; //c[5]
4   for(i=0; i<size; i++) c[i] = a[i] * b[i];
5   return c;
6 }
7 void PrintArr(int *a, int size){
8   for(int i=0; i<size; i++) cout<<a[i]<<"\t";
9   cout << "\n ";
10 }
11 void main(void){
12   int *z, x[5]={1,2,3,4,5}, y[5]={5,6,7,8,9};
13   z = ArrMul(x, y, 5);
14   cout <<"First array:\n";
15   PrintArr( x, 5 );
16   cout <<"second array:\n";
17   PrintArr( y, 5 );
18   cout <<"result array:\n";
19   PrintArr( z, 5 );
20   delete [] (z); //memory deallocated. If you
21 look carefully, we are deallocating the same
   memory that we allocated, because that memory
   was passed to variable z from variable c;
 }
```

```
First array:
1    2    3    4    5
second array:
5    6    7    8    9
result array:
5    12   21   32   45
```

- Before exiting from function `ArrMul` the address stored in `*c` returned and all the variables created in `ArrMul` is destroyed.
- The control is transferred to `main` and `z` is assigned to the address value returned by `c` of `ArrMul` (line 13).
- Then the arrays represented by `x`, `y`, and `z` is printed using function `PrintArr` (line 14-19).

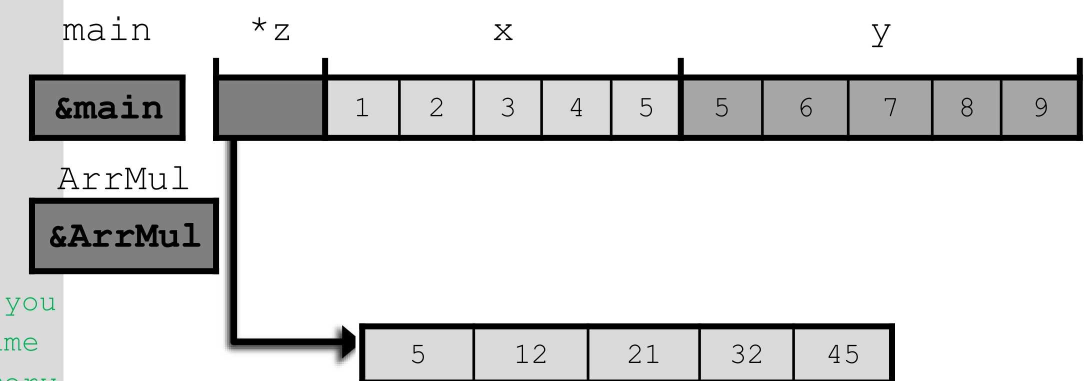


Pointer, Array & Function

```
1 //Array Multiplication
2 int *ArrMul(int a[], int b[], int size){
3   int i,*c = new int[size]; //c[5]
4   for(i=0; i<size; i++) c[i] = a[i] * b[i];
5   return c;
6 }
7 void PrintArr(int *a, int size){
8   for(int i=0; i<size; i++) cout<<a[i]<<"\t";
9   cout << "\n ";
10 }
11 void main(void){
12   int *z, x[5]={1,2,3,4,5}, y[5]={5,6,7,8,9};
13   z = ArrMul(x, y, 5);
14   cout <<"First array:\n";
15   PrintArr( x, 5 );
16   cout <<"second array:\n";
17   PrintArr( y, 5 );
18   cout <<"result array:\n";
19   PrintArr( z, 5 );
20   delete [] (z); //memory deallocated. If you
21 look carefully, we are deallocating the same
   memory that we allocated, because that memory
   was passed to variable z from variable c;
   }
```

```
First array:
1    2    3    4    5
second array:
5    6    7    8    9
result array:
5    12   21   32   45
```

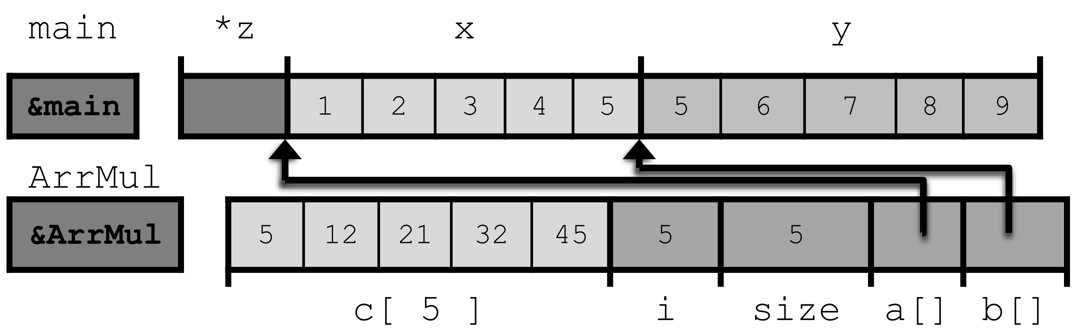
- ❑ Line 20 de-allocates the memory allocated to `*c` in function `ArrMul` (line 3) and later returned to `*z` in function `main` (line 13).
- ❑ A dynamically allocated memory must be de-allocated.



Pointer, Array & Function

```
1  //Array Multiplication
2  int *ArrMul(int a[], int b[], int size){
3    int i, c[5];
4    for(i=0; i<size; i++) c[i] = a[i] * b[i];
5    return c;
6  }
7  void PrintArr(int *a, int size){
8    for(int i=0; i<size; i++)
9  cout<<a[i]<<"\t";
10   cout << "\n ";
11 }
12 void main(void){
13   int *z,x[5]={1,2,3,4,5},y[5]={5,6,7,8,9};
14   z = ArrMul(x, y, 5);
15   cout <<"First array:\n";
16   PrintArr( x, 5 );
17   cout <<"second array:\n";
18   PrintArr( y, 5 );
19   cout <<"result array:\n";
20   PrintArr( z, 5 );
21 }
```

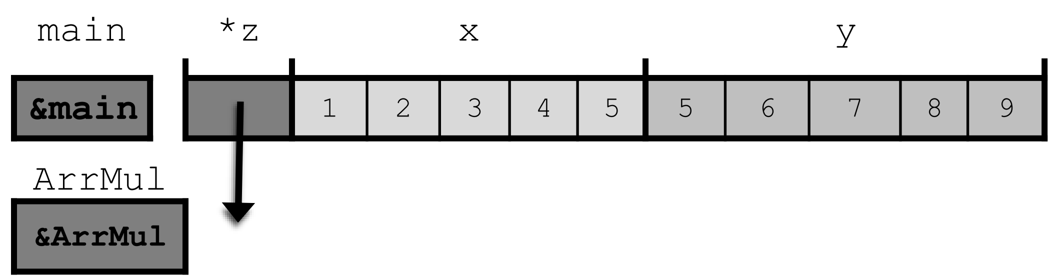
- Consider the case, if dynamic array `*c` in `ArrMul` was declared as an array `c[5]`.
- That is, instead of `int *c=new int[size]` in line 3, if it was `int c[5]`, the following would happen.
- The address represented by `c` is returned and all the variables created in `ArrMul` is destroyed and control is transferred to `main` at exit from `ArrMul`.



Pointer, Array & Function

```
1  //Array Multiplication
2  int *ArrMul(int a[], int b[], int size){
3    int i, c[5];
4    for(i=0; i<size; i++) c[i] = a[i] * b[i];
5    return c;
6  }
7  void PrintArr(int *a, int size){
8    for(int i=0; i<size; i++)
9  cout<<a[i]<<"\t";
10   cout << "\n ";
11 }
12 void main(void){
13   int *z,x[5]={1,2,3,4,5},y[5]={5,6,7,8,9};
14   z = ArrMul(x, y, 5);
15   cout <<"First array:\n";
16   PrintArr( x, 5 );
17   cout <<"second array:\n";
18   PrintArr( y, 5 );
19   cout <<"result array:\n";
20   PrintArr( z, 5 );
21 }
```

- The address represented by `c` is returned and all the variables created in `ArrMul` is destroyed and control is transferred to `main` at exit from `ArrMul`.
- `*z` is assigned to the address value returned by `c` (line 13).
- `PrintArr` finds an error when trying to print the array `z` (line 19), as the address represented by `z` has already been destroyed at the exit of the function `ArrMul`.
- So, while returning a pointer to any variable or memory area, must make sure that the returned memory area is active or not destroyed after return.



Pointers & Initialization

- Consider the statement `int *p;` which declares a pointer `p`, and like any other variable, this space will contain garbage (random numbers), because no statement like `p = &someint;` or `p = new int;` has yet been encountered which would give it a value.

- Writing a statement `int *p=2000;` is syntactically correct as `p` will point to the 2000th byte of the memory. But it might fail as byte 2000 might be being used by some other program or may be being used by some other data type variable of the same program. So such initialization or assignment must be avoided unless the address provided is guaranteed to be safe.

- There is an important difference between these definitions:

```
char amsg[] = "now is the time"; /* an array */
char *pmsg = "now is the time"; /* a pointer */
```

  - `amsg` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amsg` will always refer to the same storage and size after declaration and initialization.
  - On the other hand, `pmsg` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

Structure

Definition

- The array takes simple data types like `int`, `char` or `double` and organizes them into a linear array of elements all of the same type.

- Now, consider a record card which records *name*, *age* and *salary*. The name would have to be stored as a *string*, the age could be `int` and salary could be `float`. As this record is about one person, it would be best if they are all stored under one variable.

- At the moment the only way we can work with this collection of data is as separate variables. This isn't as convenient as a single data structure using a single name and so the C language provides *structure*.

- A *structure* is an aggregate data type built using elements of other types.

Defining Structure in C++

- In general "structure" in C++ is defined as follows:

```
struct name{
  list of component variables
};
```

Here `struct` is the key word, `name` is an identifier defining the structure name, `list of component variables` declares as much different type of variables as needed. The structure `name` works as the *new data type* defined by the *user*. Definition ends with a semicolon.

- For example, suppose we need to store a `name`, `age` and `salary` as a single structure. You would first define the new data type using:

```
struct EmployeeRecord{
   char name[5];
   int age;
   float salary;
};
```

So `EmployeeRecord` is the new user defined data type and a variable `b` of type `EmployeeRecord` can hold total 22 bytes of information [5 consecutive characters (5*2=10 bytes), followed by an integer (4 bytes), and a floating point number (8 bytes)].

Just like when we say `int` is a compiler defined data type and a variable `x` of type `int` can hold 4 bytes of integer number.

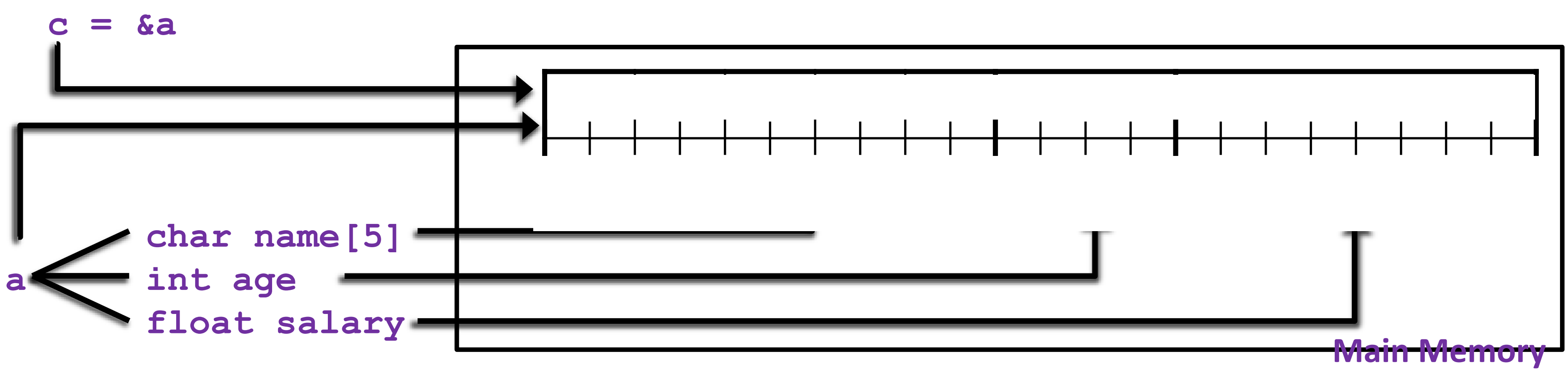
Declaring Variable of a Structure

- As we can declare variables for compiler defined data types (example: `int a, b, *c, d[50];`), we can do the same for user defined data type created using `struct`.

```
struct EmployeeRecord{
    char name[5];
    int age;
    float salary;
}a;
EmployeeRecord b, *c, d[5];
```

Variable a takes –
`5*sizeof(char)+1*sizeof(int)+1*sizeof(float)`
`= 5*2+1*4+1*8 = 22` bytes in the memory.

22 bytes will be distributed sequentially to the structure members `name`, `age`, and `salary`.

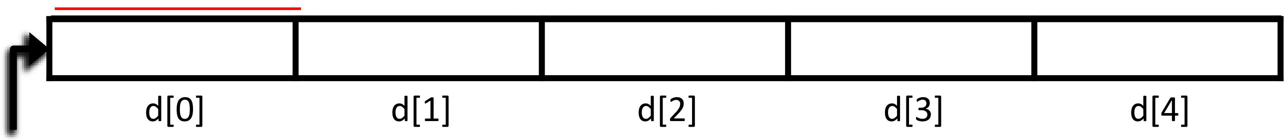


Declaring Variable of a Structure

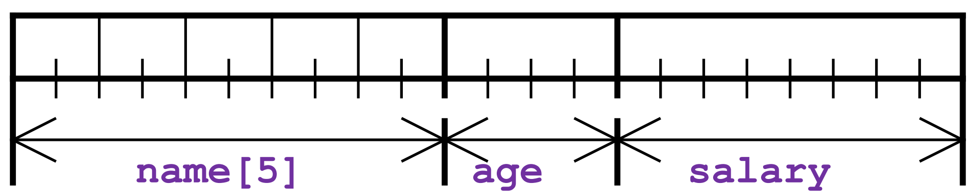
- As we can declare variables for compiler defined data types (example: `int a, b, *c, d[50];`), we can do the same for user defined data type created using `struct`.

```
struct EmployeeRecord{
   char name[5];
   int age;
   float salary;
}a;
EmployeeRecord b, *c, d[5];
```

Each index contains 22 bytes



Each 22 bytes contains the followings



Accessing Structure Member

- The dot `(.)` or combination of dash-line and greater-than sign (**->**) is used as operator to refer to members of `struct`.

```
struct EmployeeRecord{
    char name[5];
    int age;
    float salary;
};
EmployeeRecord x, y[5], *p;
x.age = 22;
x.salary = 1234.56;
strcpy(x.name, "Sam");
y[2].age = 22;
p = &x;
p->age = 22;
```

- Here, variable `x` is of type `EmployeeRecord`.
- `x.age` is of type `int`.
- `x.salary` is of type `float`.
- `x.name` is of type `char[5]`.
- `y[2].age` is of type `int` where `y[2]` is of type `EmployeeRecord` and represents the 3rd element.
- `p->age` is of type `int` where `p` is a pointer pointing to variable `x` of type `EmplyeeRecord`. Operator `(->)` is used for pointer variable of `struct` instead of `(.)`.
- `p->age` can be represented as `(*p).age` also.

- Member variables can be used in any manner appropriate for their data type.

Initializing Structure Variable

- To initialize a struct variable, follow the struct variable name with an equal sign, followed by a list of initializers enclosed in braces in sequential order of definition.

```
struct EmployeeRecord{
    char name[5];
    int age;
    float salary;
};

EmployeeRecord x = {"Sam", 22, 1234.56};
```

- `"Sam"` is copied to the member `name` referred as `x.xame`.
- `22` is copied to the member `age` referred as `x.age`.
- `1234.56` is copied to the member `salary` referred as `x.salary`.

Some Facts About Structure

- No memory (as data) is allocated for defining struct.
- Memory is allocated when its instances/variables are created.
- Hence struct stand alone is a template… and it cannot be initialized. You need to declare a variable of type struct.
- No arithmetic or logical operation is possible on the struct variables unless defined by the operator overloading. Example:

  EmployeeRecord `a, b;`

  Any expression like `a == b` or `a + b` etc. is not possible.

  But `a.age = b.age` is possible as both of these are of type `int`.

- Only assignment operation works. i.e. `a = b;`
- Call-by-value, call-by-reference, return-with-value, return-with-reference, array-as-parameter – all works with a struct variable as same as the normal variable concept using function in C++.

Nested Structure

- As any number and type of variables declared inside a structure, another structure can also be declared/defined inside another structure.
- Example 1:
  - ***AppDate*** and ***AppTime*** is defined inside the structure Appointment.
  - `dt` is a variable for the structure ***AppDate***.
  - `tm` is a variable for the structure ***AppTime***.
  - Both `dt` and `tm` is declared inside the structure ***Appointment***.

```
Example 1:
struct Appointment{
  struct AppDate{
    int day, month, year;
  }dt;

  struct AppTime{
    int minute, hour;
  }tm;

  char venue[100];
};
```

- Example 2:
  - `dob` is a variable for the structure ***DateOfBirth***.
  - `dob` is declared inside structure ***Employee***.

```
Example 2:
struct DateOfBirth{
  int day, month, year;
};

struct Employee{
  char EmpName[100];
  DateOfBirth dob;
};
```

Self-referential Structure

- Structure member cannot be instance of enclosing `struct`

- Structure member can be pointer to instance of enclosing `struct` (self-referential structure)
  - Used for linked lists, queues, stacks and trees

- Example: Every **person** may have a **child** who is also a **person**.

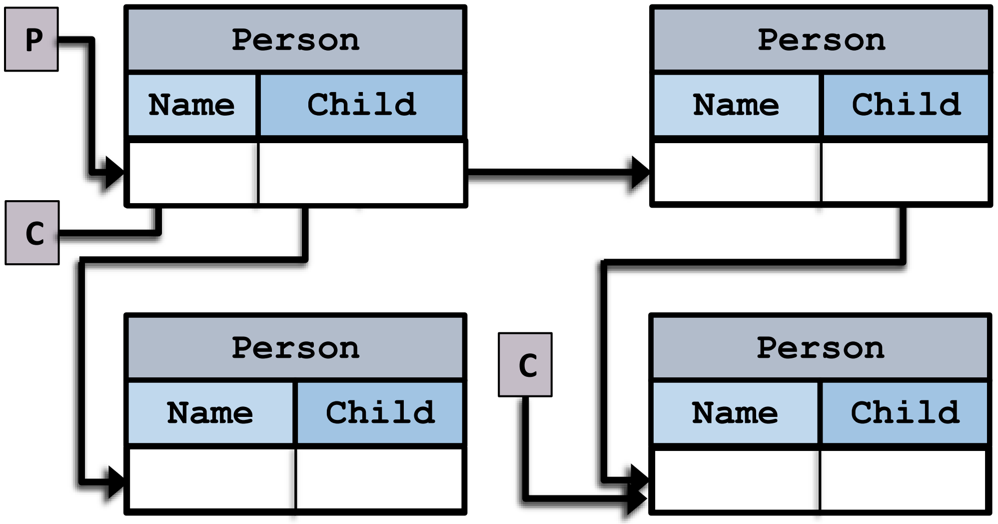
```
struct Person{
    char Name[30];
    Person *Child;
};
```

- Here, `Person` contains a pointer variable `Child` of type `Person`.

Self-referential Structure

```
struct Person{
    char Name[30];
    Person *Child;
};
Person P, *C;
strcpy(P.Name, "Arif");
C = P.Child = new Person[2];
strcpy(C[0].Name, "Sara");
C[0].Child = NULL;
strcpy(C[1].Name, "Rahim");
C = C[1].Child = new Person;
strcpy(C->Name, "Karim");
C->Child = NULL;
```

- Mr. Arif has two children – Rahim and Sara.
- Mr. Rahim has one child – Karim.
- Ms. Sara has no child.



Books

- **"Schaum's Outline of Data Structures with C++"**. By John R. Hubbard
- **"Data Structures and Program Design",** Robert L. Kruse, 3rd Edition, 1996.
- **"Data structures, algorithms and performance",** D. Wood, Addison-Wesley, 1993
- **"Advanced Data Structures",** Peter Brass, Cambridge University Press, 2008
- **"Data Structures and Algorithm Analysis",** Edition 3.2 (C++ Version), Clifford A. Shaffer, Virginia Tech, Blacksburg, VA 24061 January 2, 2012
- **"C++ Data Structures",** Nell Dale and David Teague, Jones and Bartlett Publishers, 2001.
- **"Data Structures and Algorithms with Object-Oriented Design Patterns in C++",** Bruno R. Preiss,

References

1. http://www.cplusplus.com/doc/tutorial/pointers/
2. http://www.cplusplus.com/doc/tutorial/structures/