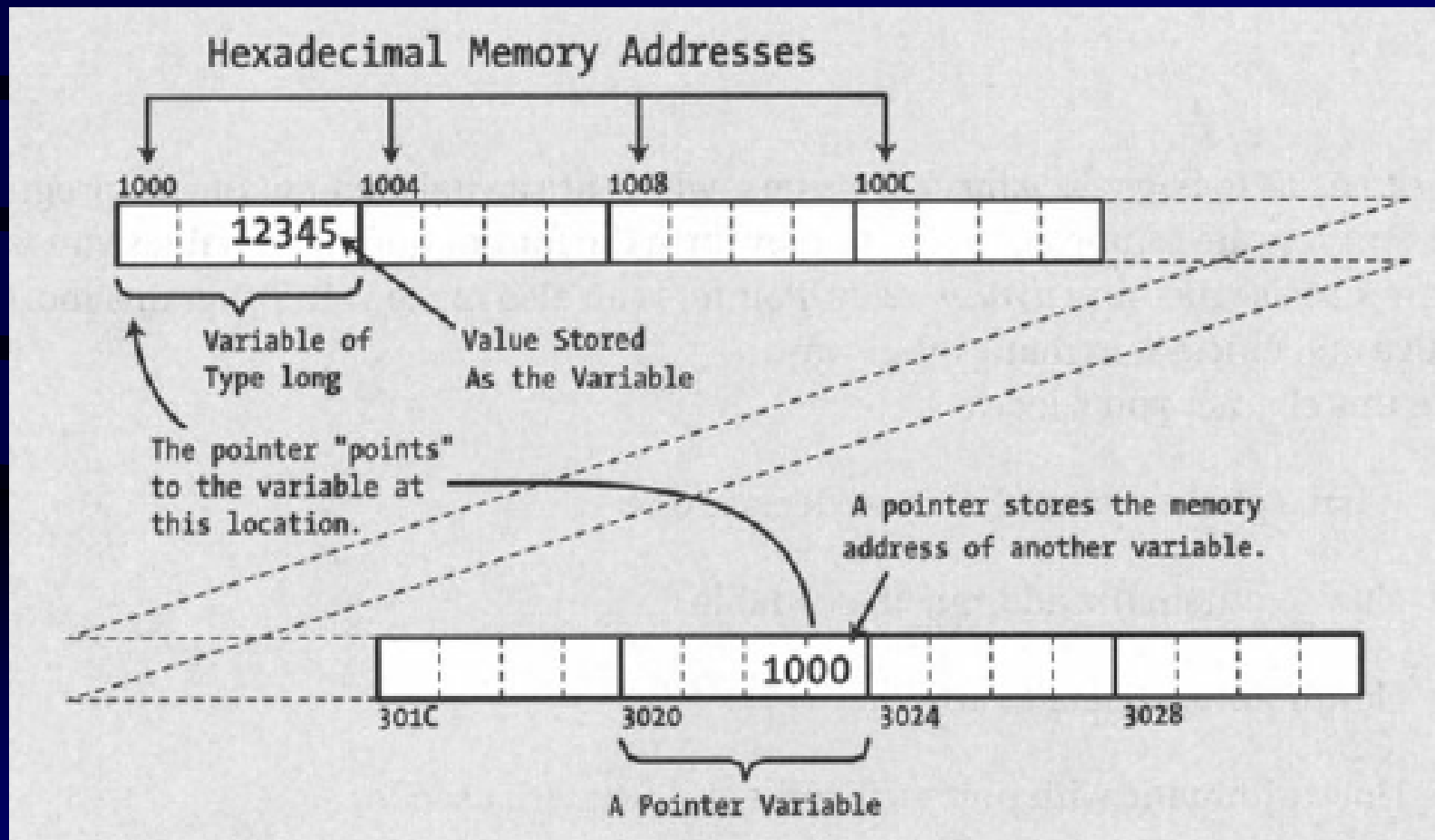


Pointers

- Declare a pointer
- Address of a variable
- Pointers and arrays
- Standard library functions
- Allocate dynamic memory space
- Release dynamic memory
- Pointer converting

What is a Pointers?



Advantages of using Pointers

- We can use pointer notation to operate on data stored in an array, which often executes faster than if we use array notation
- Use to enabling access within a function to large blocks of data which are defined outside the function
- Dynamically allocate space during program execution

Declare a pointer

- `long* plong; // long *plong;`
- `long* pn1, n2; // long *pn1, n2;`
- `double *pd; // pointer to a double value`
- `string *ps; // pointer to a sting value`

Using Pointer

- `long number = 12345L;`
- `long *pn;`
- `pn = &number;` // store address of number in pn
- `&` — **address-of** operator
- `*` — **indirection** operator; **de-reference** operator
- Program 7.1, 7.2

Initializing Pointers

- `int number = 0;`
`int* pn = &number; // initialized pointer`
- `int* pz = 0; // pointer not point to anything`
- `if (pz == 0) // if (pz == NULL) // if(!pz)`
`cout << "pz is null";`

Initializing Pointers to Type `char`

- Don't do this

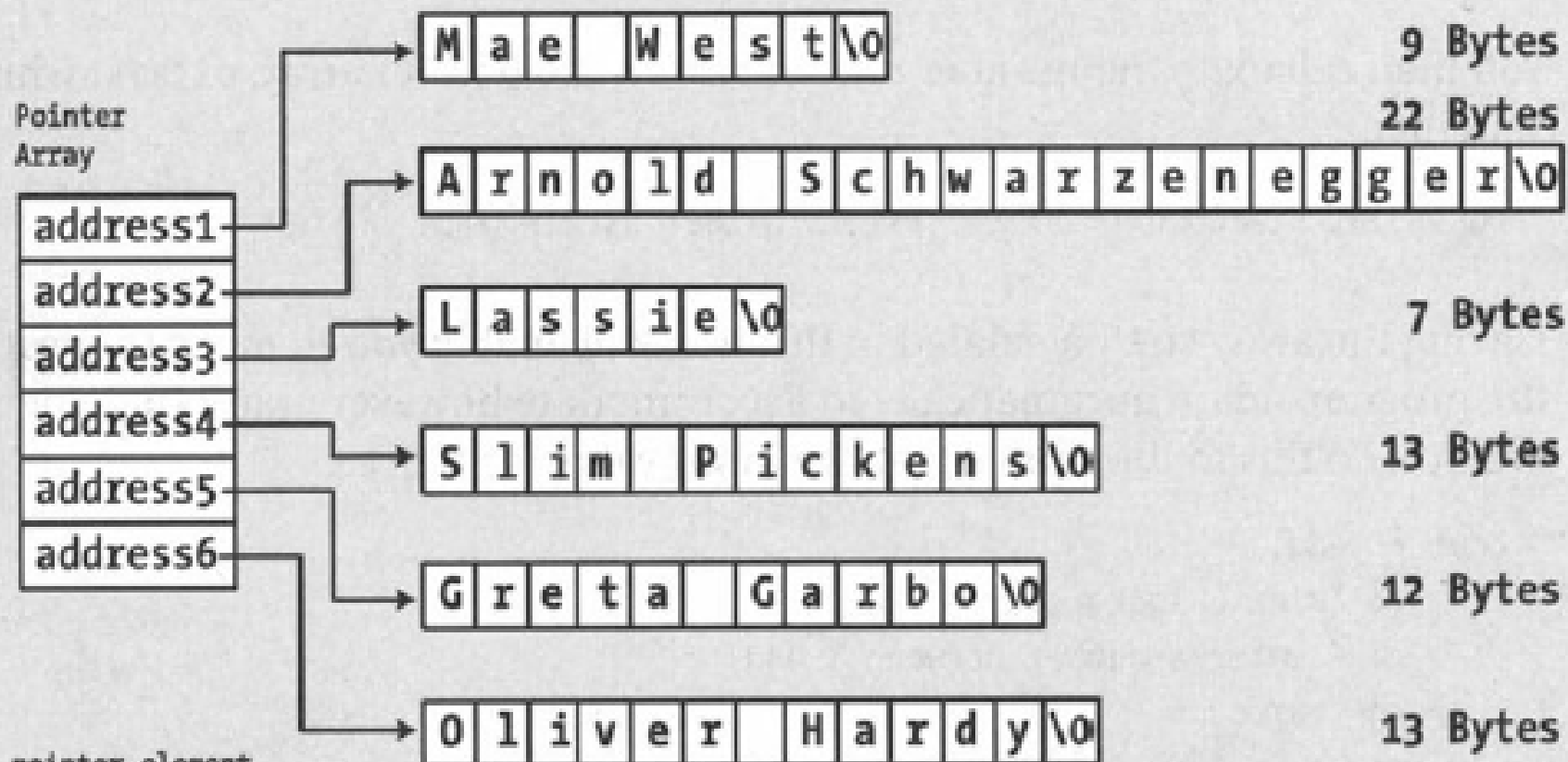
```
char* pc = "A miss is as good as a mile.";
*pc = 'x'; // execution error, program crash
```

- Right way to do

```
const char*pc = "A miss is as good as a mile.";
*pc = 'x'; // compiler error, preventing crash
```

- Program 7.3, 7.4

An Array of Pointers



Each pointer element is the same size, which is usually 4 bytes, so the array will occupy 24 bytes.

The total memory, including the pointer array, is 100 bytes.

Sorting Strings Using Pointers

- Sorting Method

```
int lowest = 0; // ascending sorting
for (int j=0; j< total_item-1; j++)
{
    lowest = j;
    for (int i=j+1; i< total_item; i++)
        if (a[i] < a[lowest])
            { lowest = i; }
    互換a[i]與a[lowest]之值;
}
```

- Program 7.5

Constant Pointers and Pointers to Constants

- A pointer to a constant
`const int value = 40;`
`const int *pi = &value;`
- A constant pointer
`int value = 40;`
`int* const pv = &value;`
- A constant pointer to a constant
`const int value = 40;`
`const int*const pv = &value;`

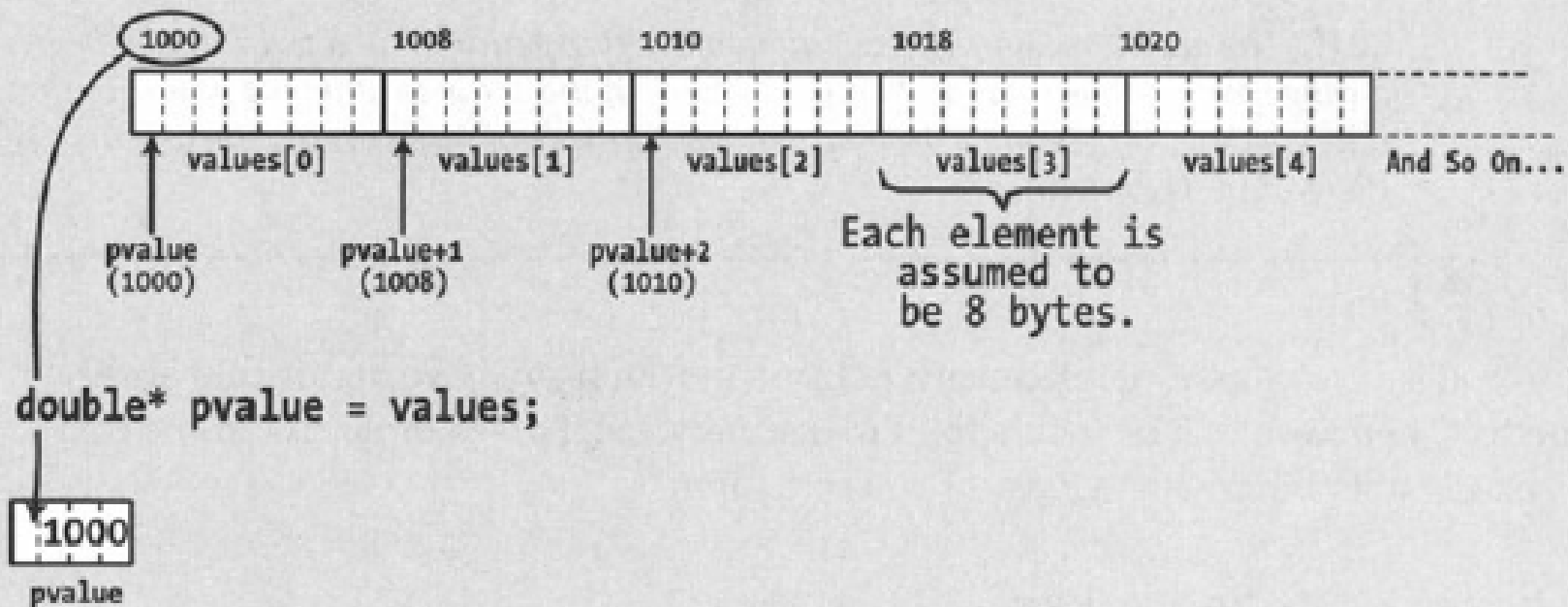
Pointers and Arrays

```
double values[10];
```

```
double *pv = values; // double* pv = &values[0];
```

`double values[10];`

Array values - Hexadecimal Memory Addresses



Precedence between * and +,

—

- Pointer priority is higher than +, -
double values[10];
double *pv = &values[3];
 *pv += 1; // values[3] +=1;
- *(pv+1) represent value[4]

Pointer Difference

- `long numbers[] = {10L, 20, 30, 40, 50, 60, 70, 80};`
- `long *pn1 = &numbers[6];`
- `long *pn2 = &numbers[1];`
- `int difference = pn1 - pn2; // 5`
- `*(pn1-3)= 100; // numbers[3]=100`
`// pn1 still point to number[6]`

Using Pointer Notation with an Array Name

```
long data[5];  
for (int i=0; i<5; i++)  
    *(data+i) = 2*(i+1); // data[i] = 2(i+1);  
for (int i=0; i<5; i++)  
    sum += *(data+i);    // sum+=data[i];
```

- Program 7.6

Using Pointers with Multidimensional Arrays

```
double beans[3][4];
```

```
double *pb = &beans[0][0];
```

```
// double *pb = beans[0];
```

```
double *pb = beans; // error
```

```
double (*pb)[4] = beans; // ok but complex
```

Multidimensional Array

```
double carrots[3][4];
```

此列是 carrots[0]

carrots[0][0]

carrots[0][1]

carrots[0][2]

carrots[0][3]

此列是 carrots[1]

carrots[1][0]

carrots[1][1]

carrots[1][2]

carrots[1][3]

此列是 carrots[2]

carrots[2][0]

carrots[2][1]

carrots[2][2]

carrots[2][3]

可用 carrots 參考整個陣列

Using Pointer Notation with Multidimensional Arrays

```
double beans[3][4];
```

```
double *pb = beans[0];
```

- $*(*(beans+i) +j)$ represents $beans[i][j]$
- Program 7.7

C-style String operations

`#include <cstring>`

`strcat()`

`strcpy()`

`strcmp()`

`strchr()`

`strlen()`

```
char name[50] = "Bing";  
char surname[] = "Crosby";
```

```
strcat(name, " ");
```

```
strcat(name, surname);
```

```
strcat(strcat(name, " "), surname);
```

Dynamic Memory Allocation

- new and delete operators

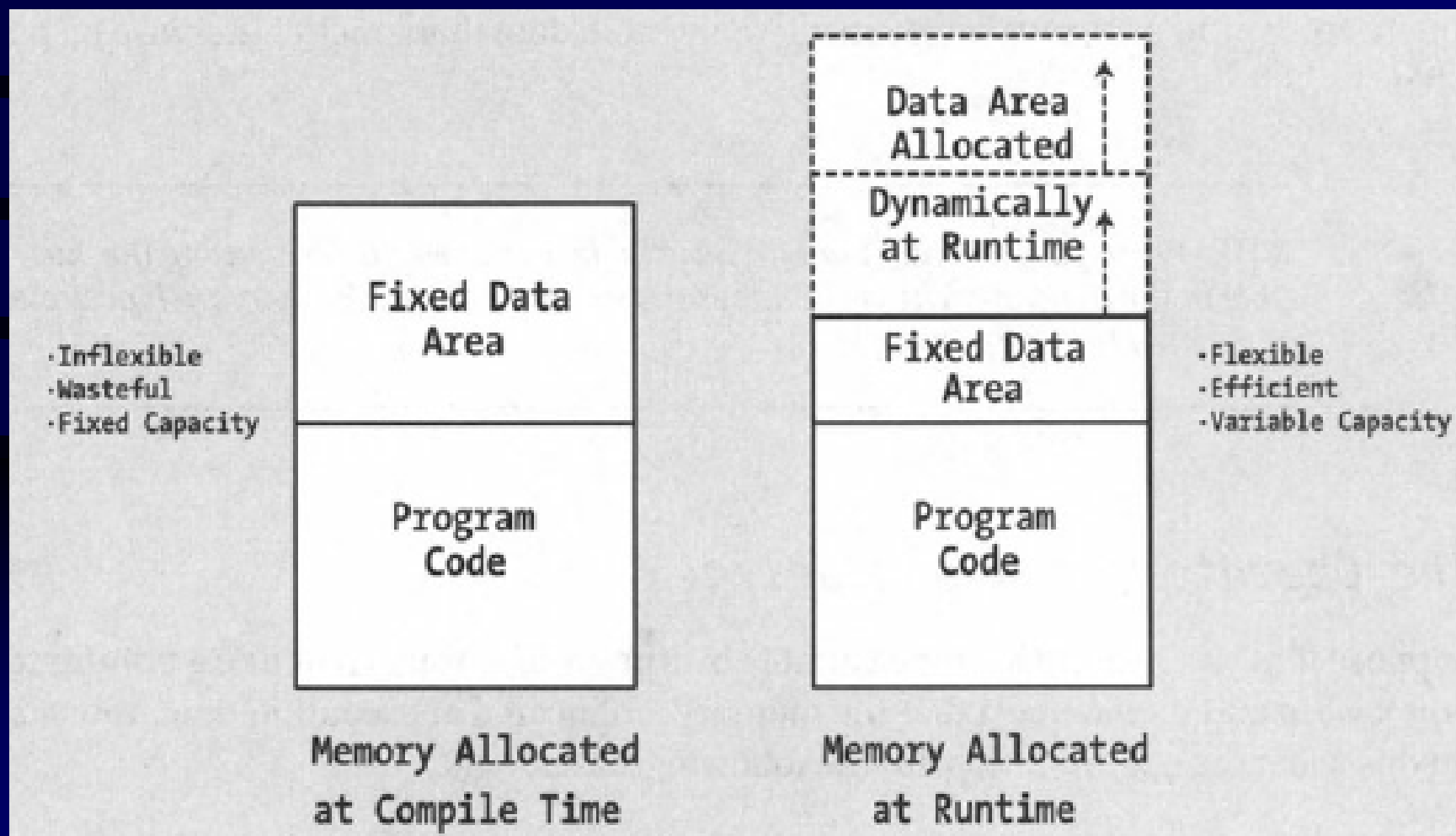
```
double *pv = 0; // null initialization
```

```
pv = new double; }  
*pv = 999.0;      } pv = new double(999.0);
```

```
delete pv; // release memory pointed to by pv
```

```
pv = 0; // reset the pointer to NULL
```

Statics versus Dynamic Memory Allocation



The free store (aka the Heap)

```
char *pc=0;
```

```
pc = new char[20];
```

```
delete [] pc; // delete array pointed to by pc
```

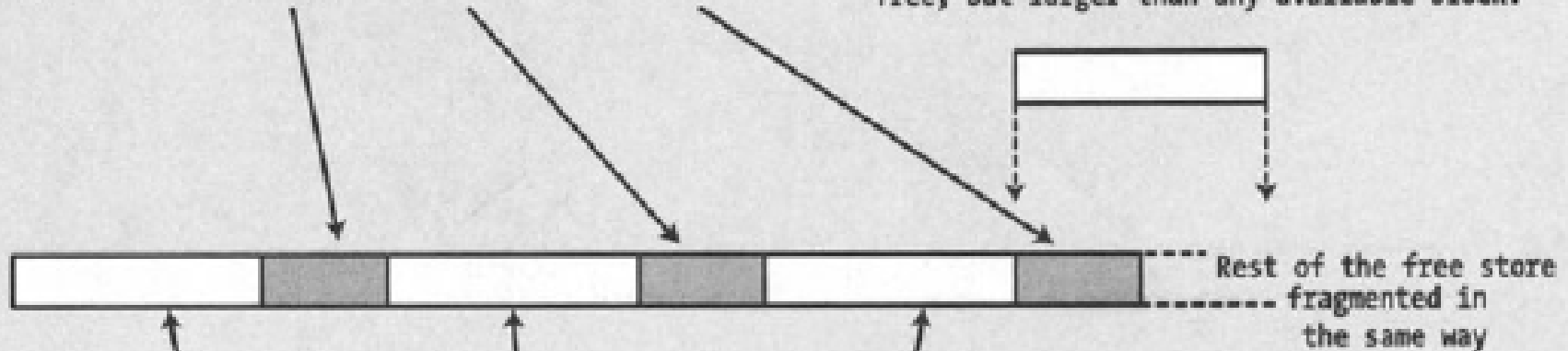
```
pc = 0;      // reset the pointer
```

- Program 7.8

Shortcomings for DMA

Memory blocks that have been released but that are all smaller than the next block required to be allocated. The total amount of memory that is free may be very large, but all in small blocks.

A new block of memory that is required is much smaller than the total memory that is free, but larger than any available block.



Memory Allocated and Still in Use
are no longer accessible.

A memory leak occurs when a block of memory is allocated in the

Multidimensional DMA

```
float **pf = new (float * [m]);
```

```
for (int j = 0; j < m; j++)
```

```
    pf[j] = new float [n];
```

```
...
```

```
for (int j = 0; j < m; j++)
```

```
    delete [] pf[j];
```

```
delete [] pf;
```

Converting Pointers

- `reinterpret_cast<pointer_type> (expression)`

```
float value = 2.5f;
```

```
float* pv = &value;
```

```
long *pnumber = reinterpret_cast<long*> (pv);
```

```
cout << *pv << *pnumber;
```

```
// 2.5 1075838976
```