22.5HV2

SOFTWARE ENGINEERING II

Pointers and dynamic memory allocation in C++

Aims

- **In this unit we will consider the following topics:**
  - **Call-by-reference functions in C.**
  - **Pointer initialisation.**
  - **Dynamic memory allocation.**
  - **The `malloc()` and `free()` functions.**
  - **The `new` and `delete` operators.**
  - **Pointers to 1-D arrays.**
  - **Pointers to 2-D arrays.**
  - **Pointers to structures.**
  - **Pointers to functions.**

Revision - pointers

- **Pointers are one of the most powerful features of the C programming language.**
  - **Pointers are also a very important part of C++.**
- **A pointer is a data type that can be used to store the *address* of a memory location where a *variable* is stored.**
- **We have already encountered pointers in the guise of call-by-reference functions.**

Call-by-reference functions in C

```
void main() {
    int a=1, b=4;
/* call */
    swap(&a,&b);
} /* end of main */
```

`i` and `j` are *pointers*,and are initialised with the *addresses* of `a` and `b`.

```
void swap(int *i,int *j) {
    int temp=*i;
    *i=*j;
    *j=temp;
    return;
} /* end of max */
```

Call-by-reference functions in C

- **The arguments of the function `swap()`, are *pointers to* `int`.**
  - **They can be used to store the *addresses* of memory locations where integers are stored.**
- **They are <u>declared</u> as being pointers to `int`, by the asterix (*) notation.**
- **This statement allocates some memory (usually 4-8 bytes), where an address can be stored.**

```
void swap(int *i,int *j) {
    int temp=*i;
    *i=*j;
    *j=temp;
    return;
} /* end of max */
```

Call-by-reference functions in C

- **This function is called using the address operator (&) to access the addresses of two integers:**

```
int a=1, b=4;
swap(&a,&b);
```

- **These *addresses* are passed by value to the function, and are used to initialise the pointers `i` and `j`.**
  - **Hence `i` and `j` will contain the addresses of `a` and `b` respectively.**
- **Knowing the addresses of `a` and `b` is not enough, we want to access the stored values.**
- **This is achieved using the *indirection operator* (*).**

```
void swap(int *i,int *j) {
    int temp=*i;
    *i=*j;
    *j=temp;
    return;
}
```

Why pointers?

- As we can see, call-by-reference functions (in C) are only possible using pointers.
- Pointers can be used to make other programming tasks easier or more efficient.
- Another example will illustrate one of the most important aspects of pointers, *dynamic memory allocation*.

Why pointers?

- **Suppose that you are writing a program to perform some image processing task.**
  - **As an image is simply a two-dimensional grid of pixels, you may choose to use an array to store the image within your program:**

```
unsigned char im[512][512];
```

- **PROBLEM: You have to decide at compile time how big to make your array `im`:**
  - **Too big and you will waste memory when processing small images.**
  - **Too small and your program will crash for large images.**

- **SOLUTION: Wait until run time to see how big an image is, and *allocate* exactly the required amount of memory.**
  - **This is called *dynamic memory allocation*, and requires pointers.**

Pointers

- To recap, we defined a pointer as being a variable that contains an address that *points* to the memory location where another variable is stored.

- We also saw how to declare a pointer using the asterix notation (`*`):

```
int  *ptr;
```

- The keyword `int`, indicates that this pointer can only be used to point to integer variables.

- This pointer will contain a random address, as we have not initialised its value. Any attempt to change the contents of this address could have disastrous consequences!.

Pointer initialisation

- For this reason, it is good practice to intitialise the pointer:

```
int  i=3, *ptr=&i;
```

- In this case, `ptr` will contain the address of the memory location where the value of integer `i` is stored.

- If we *dereference* this pointer with the indirection operator (`*`), we can change the value of `i`:

```
int i=3, *ptr=&i;
*ptr=5; // i now equals 5
```

- We have accessed the memory location where the value of `i` is stored, and placed the integer 5 there.

Pointer types

- It may seem strange at first, but a pointer is restricted to point to variables of a certain type.

- This can be better understood by considering a pointer `ptr`, initialised to point to some location in memory:

```
*ptr=57;
```

- This statement will place the value 57 in the location whose address is stored in `ptr`.

- However, the format that this value is stored as will depend on the type in the pointer declaration.
  - Possible types include `int`, `char`, `float`, `double` etc.

Pointers and arrays

- **Arrays and pointers are closely related:**

```
int  a[10], *ptr=a;
```

- **The expression `a`, represents the address of the first element in the array. Hence the pointer `ptr` has been initialised to point to the start of this array.**

- **The following statement will set the first element of the array `a` to 1:**

```
*ptr=1;
```

- **We can use the array subscript operator (`[]`) to access the other elements as follows:**

```
ptr[4]=1;
```

Pointers and arrays

- **This statement will be equivalent to the statement:**

```
a[4]=1;
```

- **As pointers are *variables*, they can be used to access different arrays:**

```
#include <iostream.h>
void main() {
   int  a[]={1,3,5,7,9},b[]={2,4,6,8,10},*ptr,i;
   char c;
   cout << "Enter odd(o) or even (e): ";
   cin >> c;
   if ( c=='o' ) ptr=a; // ptr points to a
       else      ptr=b; // ptr points to b
   for (i=0;i<5;i++)
       cout << ptr[i] << endl;
}
```

Passing arrays to functions

- When passing an array to a function, we use the syntax:

```
void display(char word[])
```

- An equivalent version is to declare the argument as a *pointer* to `char`:

```
#include <iostream.h>
void display(char *word) {
    int i, j, len=strlen(word);
    for (i=0; i<len; i++) {
        for (j=0; j<=i; j++) cout << word[j];
        cout << endl;
    }
}
void main() {
    char name[]="Heriot-Watt";
    display(name);
}
```

output

```
H
He
Her
Heri
Herio
Heriot
Heriot-
Heriot-W
Heriot-Wa
Heriot-Wat
Heriot-Watt
```

A word of caution using pointers

- Whether you use an array or pointer declaration, there is a danger that you exceed the array bounds, i.e. access a memory location outside the array:

```
int a[10], *ptr=a;
a[10] = 1;    // ERROR
ptr[11] = 1; // ERROR
```

- This is possible because C++ does not check to make sure that the element that you are attempting to access is within the allocated memory.

- Attempting to change memory locations outwith an array's bounds, could cause your program to give errors, or even crash the machine.

An alternative notation

- There is an alternative notation for accessing elements of an array using a pointer to the start of the array:

```
int a[10], *ptr=a;
ptr[4] = 1;   // OR equivalently
*(ptr+4)= 1;
```

- This notation reflects how the element is accessed:
  - The indirection operator (*) is used to access the memory location represented by the contents of the brackets.
  - The expression in the brackets equates to the address of the 5th element of the array (index 4).

Pointer arithmetic

- **The expression `(ptr+4)` represents the address of the 5th element of the array `a`.**

- **However, this expression does not use the standard arithmetic - WHY?**
  - **An address represents a memory location in <u>bytes</u>.**
  - **However, an `int` requires several bytes (4 to 8) to store in memory.**
  - **Hence, the compiler evaluates the expression `(ptr+4)` as:**

```
ptr + 4*sizeof(int)
```

  - **This ensures portability between different architectures that represent data types with different precisions.**

The alternative notation

- The previous example using the new notation is:

```
#include <iostream.h>
void main() {
   int  a[]={1,3,5,7,9},b[]={2,4,6,8,10},*ptr,i;
   char c;
   cout << "Enter odd(o) or even (e): ";
   cin >> c;
   if ( c=='o' ) ptr=a; // ptr points to a
       else      ptr=b; // ptr points to b
   for (i=0;i<5;i++)
       cout << *(ptr+i) << endl;
}
```

Pointer arithmetic

- The increment operator (++) can be used in conjunction with pointers:

```
int a[10];
int *ptr=a; // points to 1st element
ptr++;      // now points to 2nd element
```

- Remember that placing the increment operator after the variable returns the current value and increments:

```
int a[]={1,4,9,16,25};
int *ptr=a, i;
for (i=0;i<5;i++)
   cout << *ptr++ << endl;
```

Pointer arithmetic

```
#include <iostream.h>
void main() {
   int a[100], *ptr=a; // initialise pointer
   cout << "Enter +ve numbers (max 100)" << endl;
   cout << "(Terminate with a -ve number)" << endl;
   do {
      cin >> *(ptr++);
   } while (*(ptr-1)>0);
   cout << "The numbers entered were:" << endl;
   ptr=a; // reset pointer to start of array
   while ( *ptr>0 )
      cout << *ptr++ << endl;
}
```

Dynamic memory allocation

- **So far we have used pointers to point to memory locations that were allocated by *variable definitions*:**

```
int i;        // definition allocates 1 int
int a[10];    // definition allocates 10 ints
int *ptr=i;   // point to i's memory location
ptr=&a[3];    // point to address of 4th element
```

- **One of the most powerful applications of pointers, is when the memory that they are used to access is allocated *dynamically*.**
  - **Memory allocated dynamically, is not associated with a variable name - it must be accessed via a pointer.**

The `malloc()` function

- In C, the function for memory allocation is `malloc()`.

```
#include <stdio.h>
#include <malloc.h> /* must be included */
void main() {
   int *ptr, num;
   printf("Enter number of elements: ");
   scanf("%d",&num);
   ptr = (int*)malloc(num*sizeof(int));
/* rest of program */
   free(ptr);  /* deallocates memory */
}
```

- `malloc()` is used to allocate a space for an `int`, and the address is returned to `ptr`. Memory is deallocated using `free()`.

The `malloc()` function

- The function `malloc()` takes a single argument representing the number of <u>bytes</u> required.

- In this example, we wish to allocate sufficient space to store `num` integers.
  - NOTE: Different architectures will use a different number of bytes to represent an `int`. To ensure portability, we use the `sizeof()` function that returns the number of bytes for the architecture that the code is being compiled on.

```
ptr = (int*)malloc(num*sizeof(int));
```

  - If successful, `malloc()` will allocate the specified number of bytes, and returns the start address, which is assigned to `ptr`.
  - We need to cast this address as a pointer to `int` by using `(int*)`.

The `new` and `delete` operators

- **In C++, we have an alternative to `malloc()` and `free()`, namely the `new` and `delete` operators. These have the following advantages:**
  - **You don't have to include a header file as is necessary for `malloc()` and `free()`.**
  - **You don't have to use a type cast before assigning to a pointer. The `new` operator automatically returns the right kind of pointer.**
  - **Most importantly, as we shall see later, the `new` and `delete` operators have special significance when we are declaring *objects* (variables defined from classes) - namely they call special member functions called *constructors* and *destructors*.**
  - **We shall use `new` and `delete` for all our work involving classes.**

The `new` and `delete` operators

- The equivalent C++ version of the previous example is:

```
#include <iostream.h>
void main() {
   int *ptr, num;
   cout << "Enter number of elements: ";
   cin >> num;
   ptr = new[num];  // allocates memory
// rest of program
   delete [] ptr;   // deallocates memory
}
```

The `new` and `delete` operators

- The general usage of the `new` operator is :

```
pointer = new type;
```

for a single element of *type*, or ...

```
pointer = new type[number];
```

... for *number* elements of *type*.

- The corresponding uses of the `delete` operator are:

```
delete pointer;
```

```
delete [] pointer;
```

Testing for success

- There is no guarantee that the memory allocation will be successful:
  - `new` could possibly fail to allocate memory, if there is not sufficient memory available.

- If `new` is unsuccessful it will return the `NULL` pointer:
  - We can use this in a test which will exit the program will an error message if unsuccessful:

```
if ( (ptr = new int[num]) == NULL) {
   cerr << "ERROR: Cannot allocate memory!" << endl;
   return 1;
}
```

  - This procedure is more useful when allocating larger amounts of memory.

Deallocating memory

❍ **The memory that is allocated by a variable definition, is deallocated when the variable *goes out of scope*:**

```
void main() {
    int i;          // allocates memory for 1 int
    double a[10]; // allocates memory for 10 doubles
 // rest of program
} // memory reserved for i and a is deallocated
```

❍ **As the integer `i` and the array of doubles `a`, are both defined within `main()`, they *go out of scope* at the end of `main()`.**

- ❒ **Hence, the compiler deallocates or frees the memory that was reserved for them.**

Deallocating memory using `delete`

- The memory that is reserved by the `new` operator is not associated with a variable, and the compiler will not deallocate it automatically.

- Failure to deallocate memory, will lead to a memory leak:
  - Each time your program is run, it will reduce the amount of available system memory, until eventually the computer crashes!.

- Hence dynamically allocated memory must be deallocated by the `delete` operator.

  - Care must be taken to ensure that memory allocated as a number of elements is deallocated using `delete []`.

Deallocating memory using `delete`

❍ **In the previous example, we allocated `num` elements:**

```
#include <iostream.h>
int main() {
   int *ptr, num;
   cout << "Enter number of elements: ";
   cin >> num;
   if ( (ptr = new[num]) == NULL) {  // allocates memory
      cerr << "ERROR Cannot allocate memory!" << endl;
      return 1;
   }
// rest of program
   delete [] ptr;      // deallocates memory
   return 0;
}
```

- **If we used `delete ptr` instead of `delete [] ptr`, we would only deallocate the memory required for the first integer pointed to by `ptr`.**

Using dynamically allocated arrays

- Once allocated, we can use the memory as follows:

```
#include <iostream.h>
int main() {
   int *ptr, num, i;
   cout << "Enter number of elements: ";
   cin >> num;
   if ( (ptr = new[num]) == NULL) {
      cerr << "ERROR Cannot allocate memory!" << endl;
      return 1;
   }
   for (i=0;i<num;i++) {
      cout << "Enter value " << i+1 << ": ";
      cin >> ptr[i];
   }
// rest of program
   delete [] ptr;
   return 0;
}
```

Comparison with arrays

- Arrays use static memory allocation, i.e. the required number of elements must be specified by compile time.
  - This is OK when the size of the array is known and is fixed for all time.
  - When the array size is variable, the problem is what size to make the array: too big leads to a waste of memory, too small retricts your program's application.
- The major advantage of pointers and dynamic memory allocation, is that we can wait until run-time to see how much memory is required, and allocate *exactly* the required amount of memory.
- Programs that use dynamic memory allocation are more flexible and efficient that programs that use arrays.

Image processing example

- Consider the situation where you have to write a program to *invert* a black and white image.

- Most monochrome images represent each pixel by a grey-level value in the range of 0 (black) to 255 (white).
  - We can store an image in a 2D array of `unsigned char`'s.
  - To invert the image, we simply subtract each pixel grey-level from the value 255.

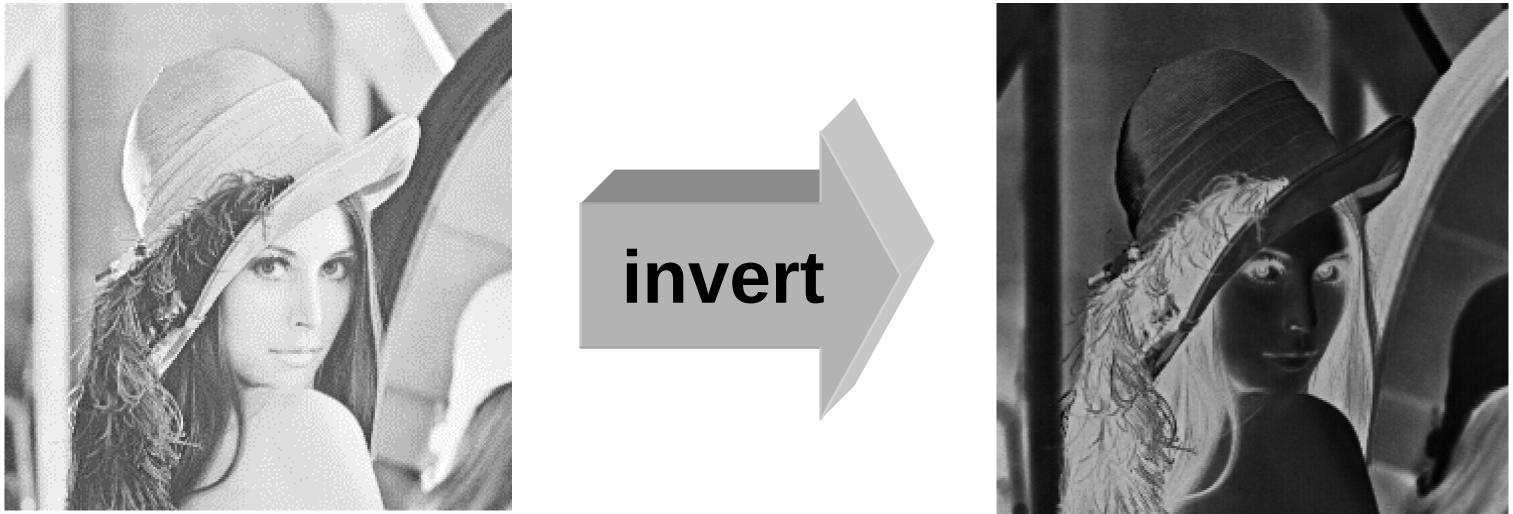


Image processing example: Arrays

- The code using an array for the image, would be:

```
#include <iostream.h>
int main() {
   const int MAX_IM_SIZE=512; // maximum image size
   unsigned char im[MAX_IM_SIZE][MAX_IM_SIZE];
   int row,cols,rows,cols;
 // read in image size (rows,cols)
   if ( rows>MAX_IM_SIZE || cols>MAX_IM_SIZE ) {
       cerr << "ERROR: Image too large!" << endl; return 1; }
 // read in image
   for (row=0;row<rows;row++)
       for (col=0;col<cols;col++)
          im[row][col] = 255 - im[row][col];
 // write out image
}
```

- This code would work for images of size up to 512 by 512.

Image processing example: Pointers

❍ The code using *dynamic memory allocation* is:

```
#include <iostream.h>
int main() {
   unsigned char *im;
   int row,cols,rows,cols;
 // read in image size (rows,cols)
   if ((im=new unsigned char[rows*cols]) == NULL) {
       cerr << "ERROR: Image too large!" << endl; return 1; }
 // read in image
   for (row=0;row<rows;row++)
       for (col=0;col<cols;col++)
          im[row*cols+col] = 255 - im[row*cols+col];
 // write out image
   delete [] im; // deallocate memory
   return 0;
}
```

Image processing example: Pointers

- This code would work for images of any aspect ratio, up to a size that can be accomodated by the available memory.

- This example illustrates the flexibility of using dynamic memory allocation.
  - The code will operate with whatever memory is available.
  - Hence, if the code is running on a machined equipped with a large amount of memory, then larger images can be processed.

- A slight disadvantage of this approach, is the less intuitive way of accessing 2D arrays.

Pointers to 2D arrays

- **For 1D arrays, we can *dereference* the pointer using the same indexing notation (`[]`) as a 1D array:**

```
int a[10], *ptr=a;
ptr[4] = 1;
```

- **Arrays of dimensions greater than one, are actually stored in memory as one dimensional arrays.**
  - **Using 2D arrays we are allowed to use the double indexing notation:**

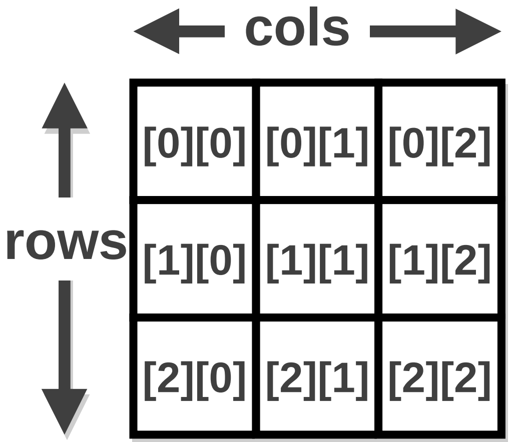
```
im[row][col]
```

  - **When using pointers, we have to use the 1D notation:**

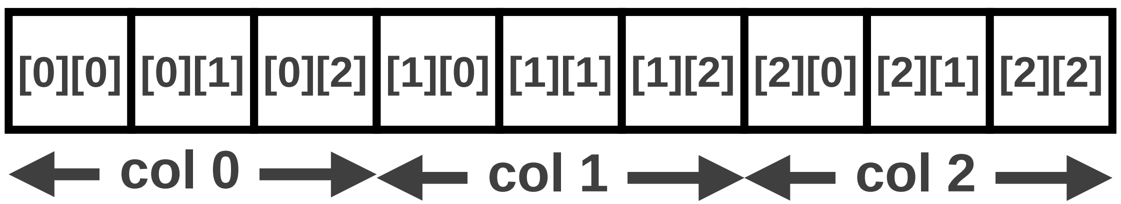
```
im[row*cols+col]
```

Pointers to 2D arrays

❍ **This notation can be better understood by considering an image of 3 rows, by 3 columns:**



Array representation



Actual format in memory

❍ **Hence, to access a particular element, we use the notation:**

```
im[row*cols+col]
```

Images as function arguments

❍ **A function to invert an image would be:**

```
void inv(unsigned char im[][MAX_IM_SIZE],int rows,int cols) {
   int row,col;
   for (row=0;row<cols;row++)
      for (col=0;col<cols;col++)
         im[row][col] = 255 - im[row][col];
}
```

- ❒ **Using 2-D arrays to store the image, OR:**

```
void inv(unsigned char *im, int rows,int cols) {
   int row,col;
   for (row=0;row<cols;row++)
      for (col=0;col<cols;col++)
         im[row*cols+col] = 255 - im[row*cols+col];
}
```

- ❒ **Using dynamic memory allocation.**

Pointers to structures

- **Pointers can be declared for any type, including structures.**
  - **This presents a slight notational problem:**
  - **Let us declare a variable `a` of type `rational`, initialise its members, and declare a pointer to `rational`, initialised to point to `a`:**

```
rational a={22,7}, *ptr=&a;
```

  - **We can access the members of `a` using its pointer `ptr` as follows:**

```
(*ptr).num
```

  - **An easier notation is to use the `->` operator:**

```
ptr->num
```

Pointers to structures

```
#include <iostream.h>
#include <string.h>
struct employee {
  char name[30];
  int  wage;
};
void main()
{
  employee *labourer = new employee;
  strcpy(labourer->name,"Bill Gates");
  labourer->wage = 100;
  cout << labourer->name << endl;
  cout << labourer->wage << endl;
  delete labourer;
}
```

Pointers to functions

- It is possible to declare a pointer to a function:
  - Allows run-time selection of functions:

```
#include <iostream.h>
void function1(void) {
   cout << "function 1" << endl;
}
void function2(void) {
   cout << "function 2" << endl;
}
void main() {
   int i;
   void (*funptr)(void); // declaration
   cout << "Function 1 or Function 2: ";
   cin >> i;
   if (i==1) funptr=&function1;
   else      funptr=&function2;
   (*funptr)();              // function call
}
```

Pointer to functions

- **<u>EXAMPLE:</u> The C++ standard library `qsort()` function.**

- **C++ provides a function to perform a quick sort on an array of any type of variable.**

  - **<u>PROBLEM</u>: this function needs to know how to compare these elements in order to sort them.**

  - **<u>SOLUTION</u>: the user writes their own function and passes a pointer to this function as one of the arguments of `qsort()`.**

- **e.g. The football league example:**
  - **A league of teams could be held in an array, and sorted into correct position, by writing a function that sorts by points and splits ties by goal difference.**

Summary

- A "pointer to `int`" is a variable that can represent the ***address*** of the memory location where an integer is stored.
- Pointers may be used to access arrays of data.
- An important application of pointers is in dynamic memory allocation.
  - Dynamic memory allocation is a more flexible and efficient way of representing arrays of data.
- In C++ the `new` and `delete` operators are used in preference to `malloc()` and `free()`.
- Care must be taken to ensure that dynamically allocated memory is deallocated before program termination.

Summary

- The –> operator may be used to access the members of structures via a pointer.
- It is possible to declare pointers to functions, which would allow the run time selection of functions.