DATA TYPES IN C++ - PART 2

Compound Types :

(Arrays, Strings, Pointers and Structure)

Arrays:

An array is data form that can hold several values, all of one type.

Ex – Array can hold 100 type *int* numbers. 7 *short* values that represent days in a week or 12 *short* values that represent months in a year.

To create an array, you use declaration statement. Array declaration should indicate three things:

1. The type of value to be stored in each element
2. The name of the array
3. The number of elements in the array.

Ex-

short months[12]; // creates an array of 12 short

General Form :

typeName arrayName[arraySize];

arraySize must be an integer constant(12), or a constant expression (5*sizeof(int)), for which all values are known at the time compilation takes place. You can use the *new* operator to remove that restriction.

An array is called compound type because it is built from some other type

An array always has to be some particular type. There is no generalized array type. Instead, there are many specific array types, such as array of *char, double* or *long*.

short students[100];

The type of students is not “array”; rather, it is array of “short”.

This emphasizes the students array is built from *short* type.

You can access any element in an array by using their **index** or **subscript**.

Always remember numbering starts with 0.

So, the last element in any array is one less than the size of array.

Note :

The compiler doesn’t check to see if you use a valid index or not. For instance, the compiler won’t complain if you assign a value to the non-existent element (students[504]). But it can cause problems when the program runs, possibly corrupting data or code, possibly causing the program to abort. So make sure that your program uses only valid subscript values.

Let’s check all these in program – ArrayOne.cpp

Initialization rules for arrays :

You can use the initialization form only when defining the array. You cannot use it later, and you cannot assign one array wholesale to another.

```
int students[5] = {78, 89, 54, 68, 98}; //allowed
int students2[5]; //allowed
students2[5] = {78, 78, 45, 99, 62}; // not allowed
students2 = students; //not allowed
```

You can use subscript or index and assign values to the elements of the array individually.

<u>If you partially initialize an array, the compiler set the remaining elements to zero</u>.

```
float incomes[3] = [4578.45, 7895421.78];
```

So, it’s easy to initialize all the array elements to zero.

```
long totalStudents[1000] = {0}; // all value will be zero
long totalStudents[1000] = {1}; //first element is 1 rest is still 0.
```

<u>If you leave the square brackets ([]) empty when you initialize an array, the C++ compiler counts the elements for you .</u>

```
short importantChapter[] = {4, 7,8, 12};
```

The compiler makes this array of four elements.

But letting the compiler count is poor practice, because its count can be different from what you think it should be.

This approach can be a safe one for initializing a character array to a String.

If you don’t know the size of an array you can find it by using this technique.

```
short importantChapter[] = {4, 7,8, 12};
int numElements = sizeof importantChapter / sizeof (short);
```

C++ 11 Array Initialization :

You can drop the = sign when initializing an array:

```
float nums[3] {32.45, 54.67, 76}; //valid in c++11
```

You can use empty braces to set all the elements to 0.

```
int nums [10] {}; // set all the elments to 0
or int nums[10] = {}; // set all the elments to 0
```

List-initialization protects against narrowing:

```
long num[5] = {45, 78, 45.0}; // not allowed because converting from a
        floating – point number to an integer type is narrowing even if the
        floating-point value has only zeros after the decimal point.
```

char[4] {‘a’,’m’, 2323432, ‘\0’}; // not allowed because 3rd value of out of range for char character.

char[4] {‘a’, ‘m’, 124, ‘\0’}; // valid because 3rd value is in the range of char character (-127 to 128).

The C++ Standard Template Library (STL) provides an alternative to array called the vector template class, and C++11 adds an array template class. These alternative are more sophisticated and flexible than the built-in array composite type. We’ll discuss it later.

String :

String is a series of characters stored in consecutive bytes of memory.

C++ has two ways to dealing with strings:

1. Token from C, often called C-style string
2. String class library (alternative method)

The idea of a series of character stored in consecutive bytes implies that you can store a string in an array of char, with each character kept in its own way to store text information.

E.g Message to the user (“enter a number!”)

Or response from the user (“9”)

C-style have a special feature: The last character of every string is the *null character*. This character written as \0, is the character with ASCII code 0, and it serves to mark the string’s end.

e.g

char book[4] = {‘b’, ‘o’, ‘o’, ’k’} ; // This is not a string

char book2[4] = {‘b’, ‘a’, ‘t’, ‘\0’}; //This is a string

Both of these are arrays of char, but only second is a string. The null character plays fundamental role in C-style string. Because when the *cout* detects the null character it stops.

In the first example, cout prints the eight letter in the array and then keeps marching through *byte-to-byte,* interpreting each byte as a character to print, until it reaches a null character.

Because null character, which are the bytes set to zero, tend to be common in memory, the damage is usually contained quickly; nonetheless, you should not treat nonstring character arrays as string.

<u>There is a better way to initialize a character array to a string. Just use a quoted string, called a string constant or constant literal.</u>

char name[11] = “A R Danish”; // \0 is understood

char name2[] = “nerd”; //let the compiler count

Quoted string always include the terminating null character implicitly, so you don’t have to spell it out. Also the various C++ facilities for reading a string from keyboard input into a char array automatically add the null character for you.

You should make sure the array is large enough to hold all the character of the string, including the null character. Initializing a character array with a string constant is one case where it may be safer to let the compiler count the number of elements for you. There is no harm, other than wasted space, in making an array larger than the string.

It’s because functions that work with strings are guided by the location of the null character, not by the size of the array.

C++ imposes no limits on the length of a String.

Note :

A string constant (with double quotes) is not interchangeable with a character constant (with single quotes).

char char1 = ‘A’; // fine either you write 65 or A

char char2 = “A”; // illegal type mismatch

“A” is not a character constant; it represents the string consisting of two characters, the A and \0 characters. Even worse “A” actually represents the memory address at which the string is stored.

Because an address is a separate type in C++, a C++ compiler won’t allow this sort of nonsense.

Concatenating String Laterals:

Sometimes a strings may be too long to conveniently fit into one line of code.

C++ enables you to add two quoted strings into one.

Any two string constants separated only by whitespaces (spaces, tabs, and newlines) are automatically joined into one.

E.g.

```
cout << “It’s always a good day” “ when you’re at home.\n”;

cout << “”It’s always a good day when you’re at home.\n”;

cout << “It’s always a good d”

“ay when you’re at”

“ home.\n”;
```

All three are equivalent.

The join doesn’t add any spaces to the joined strings. The first character of the second strings immediately follows the last character, not counting \0, of the first string. The \0 character from the first string is replaced by the first character of the second string.

Check the program – StringsOne.cpp

Using Strings in an array:

First let's check program in IDE.

StringTwo.cpp

The problem is how *cin* determines when you've finished entering a string. You can't enter the null character from the keyboard.

The cin technique is to use whitespaces – (Tab, spaces and new lines)- to delineate a string.

Another problem, it offers no protection against placing a 30-character string in a 20-character array.

Reading String Input a Line at a time:

Reading string input a word at a time is often not the most desirable choice. E.g city – San Francisco, name – A R Danish, etc,.

So, you need a line-oriented method instead of a word-oriented method.

Cin has some line-oriented class member functions – getline() and get(). Both read an entire input line. However, getline () then discards the newline character the newline character, whereas get() leaves it in the input queue.

getline () :

You invoke this method by using – cin.getline() as a function call.

It takes two arguments. The first arguments is the name of the target (variable), and the second argument is a limit on the number of characters can be read (If this limit is, say 20, the function reads no more than 19 characters, leaving room to automatically add the null character at the end).

E.g cin.getline(name, 20);

The getline() member function also has third optional argument, we'll discuss it later.

See the program – StringsThree.cpp

get() :

It comes in several variations. One variant works much like getline(). It takes the same argument, interprets the same way, and reads to the end of the line. But rather than read and discard the newline character, get() leaves that character in the input queue.

e.g.

cin.get(name, arrSize) ;

cin.get (favFood, arrSize); // a problem

It doesn't replace the enter key with \0(null) character so it reads enter key (\0) as the second string. Thus, get() concludes that it's reached the end of line without having found anything to read. Without help, get() just can’t get past that newline character.

Fortunately, there is help in the form of a variation of get().The call cin.get() (with no arguments) reads the single next character, even if it is a newline, so you can use it to dispose of the newline character and prepare for the next line of input.

E.g.

cin.get(name, arrSize) ; //read first line

cin.get(); // read newline

cin.get (favFood, arrSize); // read second line

Another way to use get() is concatenate(), or join the two class member function, as follows :

cin.get(name, arrSize).get(); // concatenate member functions

Here, cin.get(name, arrSize) returns the cin object, which is then used as the object that invokes the get() function.

Similarly,

cin.getline(name, arrSize).getline(favFood, arrSize);

This statement reads two consecutive input lines into the arrays name and favFood.

See the programs – StringsFour.cpp

Why uses get() instead of getline() ?

1. Older implementations may not have getline().
2. get() let you be a bit more careful.

   Suppose, you used get() to read a line into an array. How can you tell if you read the whole line rather than stopped because the array was filled? Look at the next input character. If it's a newline character then the whole line was read. If it is not a newline character, then there is still more input on that line.

3. getline() is a little simpler to use, but get() makes error checking simpler. You can use either one to read an input; just keep these things in mind.

Empty lines and other problems :

After get() (not getline()) reads an empty line, it sets something called *failbit*. The implications of this act are that further input is blocked, but you can restore input with the following command.

cin.clear();

Another potential problem is that the input string could be longer than the allocated space. If the input line is longer than the number of characters specified, both get() and getline() leave the remaining characters in the input queue. However, getline() additionally sets the failbit and turn off further input.

Mixing Strings And Numeric Input :

Mixing numeric digit with line-oriented string input can cause problems.

Let's see the program – StringsFive.cpp

You never get opportunity to enter the place. The problem is that when cin reads the year, it leaves the newline generated by the Enter Key in the input queue. Then cin.getline() reads the newline as an empty line and assign a null string to the place array.

The fix is to read and discard the newline before reading the place. This can be done by in several ways by using get() with a char argument or with no argument.

```
cin >> yyyy;

cin.get();
```

or you can concatenate

```
(cin >> yyyy) .get();
```

It'll work fine. See the programs – StringsSix.cpp

String Class :

The ISO/ANSI C++98 Standard expanded the C++ library by adding a string class. So, instead of using a character array to hold a string, you can use type string variable. Also, the string class is simpler to use than the array and also provides a truer representation of a string as a type.

To use a string class in your program, you must have to include the string header file. The string class is a part of the std namespace, so you have to provide the using directive or declaration or else refer to the class as std::string. The class definition hides the array nature of a string and lets you treat a string much like ordinary variable.

See the program - StringsSeven.cpp

You can use the string object in same ways as character array too.

The main difference between string object and character array is that you declare a string object as a simple variable not as an array.

The class design allows the program to handle the sizing automatically. For instance, the declaration for str1 is creates a string of length zero, but program automatically resizes when it reads the input.

cin >> str1; // str1 resized to fit input

this makes using a string object both more convenient and safer than using an array. Conceptually, one thinks of an array of char as a collection of char storage units used to store a string but of a string class variable as a single entity representing the string.

C++ 11 String Initialization :

C++ 11 enables list-initialization for C-style strings and string objects:

char name1[] = {“A R Danish”};

char name2[] {“Hannah Baker”};

string name3 = {“Bryce Walker”};

string name4 {“Clay Jenson”};

Assignment, Concatenations and Appending :

The string class makes operation simpler.

You can’t simply assign one array to another. But you can assign one string object to another.

Considering the program StringsSeven.cpp.

char1 = char2; // Invalid, no array assignment

str1 = str2; // Valid, object assignment

The string class simplifies combining strings.

You can use a + operator to add two strings objects together and the += operator to tack on a string to the end of existing string object.

e.g.

string str3;

str3 = str1 + str2 ; // assign str3 the joined strings

str1 += str2 ; // add str2 to the end of str1

You can add and append C-style strings as well as string objects to a string objects.

See the programs – StringsEight.cpp

Escape sequence \” represents a double quotation that is used as a literal character rather than as marking the limits of a string.

More string class operations :

Before C++ programmers also need to assign strings. For C-style strings, they used functions from the C library for these tasks.

The cstring header file (formerly string.h) supports these functions. Here you can use the strcpy() function to copy a string to a character array, and you can use the strcat() function to append a string to a character array.

e.g.

strcpy(char1, char2); // copy char2 to char1

strcat (char1, char2); // append content of char2 to char1

see the programs : StringsNine.cpp

suppose in string objects :

s3 = s1 + s2;

Appending is way too simple but in C style you’ve to write the following codes.

strcpy(char3, char1);

strcpy(char3, char2);

Also, with arrays there is always the designation array being too small to hold the information. E.g.

char char1[10] = “India”;

strcat(char1, “ is the best place to live.”);

Here, strcat would attempt to copy all the characters into the char1 array, thus overrunning adjacent memory.

This might cause the problem to abort, or the program might continue running but with corrupted data (You can clearly the question mark when running the program StringsNine.cpp with array size 10).

The string class automatically resizes as necessary, avoid this problem.

The C library does provide cousins to strcat() and strcpy(), called **strncat()** and **strncpy()**, that work more safely by taking a third argument to indicate the maximum size of the target array, but using them adds another layer of complexity.

When finding the size of the array the syntax is different from each another but both return the number of characters in the string. str1.size() indicates str1 is an object and size() is a class method. A method is a function that can be invoked only by an object belonging to the same class as the method.

So, C functions uses a function argument to identify which string to use, and the C++ string class objects use the object name and the dot operator to indicate which string to use.

More on string Class I/O :

Reading a line at a time instead of word at a time uses a different syntax.

See the program – StringsTen.cpp

After running the program the program says the length of the string in the array char1 before input is 1, which is smaller than the size of the array. Two things are going on here. First, The contents of an uninitialized array are undefined. Second, The strlen() function works by starting at the first element in the array and counting bytes until it reaches a null character. Here, first null character appears just at 1 so it stops. But the first null character in uninitialized data is essentially random, so you very well could get a different numeric result using this program.

cin.getline(char1, 20);

Here, getline() function is a class method for the istream class.

getline(cin, str);

This getline() is not a class method.

So why is one getline() an istream class method and the other getline() not ?

The istream class was part of C++ long before the string class was added. So the istream design recognizes basic C++ types such as double, int, but it is ignorant of the string type. Therefore, there are istream class for processing double, int, and other basic types, but there are no istream class methods for processing string objects.

So, if there is no istream class methods for processing string objects, you might wonder why this code works.

Cin >> x; // read a value into a basic c++ types

This code uses a member function of the istream class.

cin >> str; // read a word into the str string object

But the string class equivalent uses a friend function of the string class.

We'll discuss it later until then forget about inner processing and use cin and cout for string objects.

Other forms of String Literals :

C++ has wchar_t type in addition to char. And C++11 added char16_t and char32_t types. It's possible to create arrays of these types and string literals of these types. C++ uses the L, u and U prefixes, for string literals of these types.

```
wchar_t title[] = L"Katherine Langford";  // w char string
char16_t name[] = u"Clay Jhonson";    // char_16 string
char32_t car[] = U"Hannah Baker"; // char_32 string
```

C++11 also supports an encoding scheme for Unicode characters called UTF-8.In this scheme a given character may be stored in anywhere from one 8-bit unit,or octet,to four 8-bit units,depending on the numeric value.C++ uses the u8 prefix to indicate string literals of that type.

Raw Strings :

In raw string, character stands for themselves. .For example,the sequence \n is not interpreted as representing the newline character;instead,it is two ordinary characters,a backslash and an n,and it would display as those two characters onscreen.

You can “ instead of \”.

Raw strings uses “(and)” as delimiters, and they use **R** prefix to identify them as raw strings.

cout << R”(“A R” uses “\n” instead of endl.)” << ‘\n’;

Output :

“A R” uses \n instead of endl.

In standard string literal equivalent :

cout << “\”A R\” uses \”\\n\” instead of endl.” << ‘\n’;

Here we had to use \\ to display \ because a single \ is interpreted as the first character of an escape sequence.

If you press the Enter or Return key while typing a raw string, that not only moves the cursor to the next line on screen, it also places a carriage return character in the raw string.

But what if you want to print like this in raw string : “(Who wouldn’t?)”, she whispered.

Raw string syntax allow you to place additional character between the opening “ and (and) and “ (Same additional characters must appear between final). For example, raw string beginning with R”+*(must terminate with)+*”.

So,

```
cout << R”+*(“(Who wouldn’t?)”, she whispered.)+*” << endl;
```

Output :

```
“(Who wouldn’t?)”, she whispered.
```

You can use any of the members of the basic character set as part of the delimiter other than the space, the left parenthesis, the right parenthesis, the backslash, and control characters such as a tab or a newline.

Structures :

A structure is more versatile data from than an array because a single structure can hold items of more than one data type.

You can use array of structures to for more complex data too.

A structure is a user-definable type. First you define the type then you can create variables of that type.

Thus, creating a structure is a two-part process. First, you define a structure description that describes and labels the different types of data that can be stored in a structure. Then you can create structure variables, or, more generally, structure data objects, that follow the description's plan.

Here is an example of structure code:

```
struct personType // structure declaration

{

        char name[20];

        int age;

        float income;

};
```

The keyword struct indicates that the code defines the layout for a structure. The identifier persons is the name, or tag, for this form. Next, between braces are the list of data types to be held in the structure. Each list item is a declaration statement. You can use any of the C++ types here, including arrays and other structures.

Each individual item in the list is called a structure member, so the inflatable structure has three members.

After you have defined the structure, you can create variables of that type:

```
personType name1; // name is a structure variable of type personType
personType age1; // type personType variable
personType income1; // type personType variable
```

C++ allows you to drop the keyword struct. In C, you have to struct when declaring a variable.

You can use the membership operator(.), to access individual members. For example, age1.income refers to the income member of the structure.

See the program – StructOne.cpp

You could either place the declaration inside the main(), function, just after the opening braces or you could like I did in program.

When a declaration occurs outside any function, it's called an external declaration. .The external declaration can be used by all the functions following it, whereas the internal declaration can be used only by the function in which the declaration is found. Most often we want external structure declaration.

The program places one value per line, but you can place them all on the same line. Just remember to separate items with commas:

```
personType person1 = {“A R Danish”, 20, 45000.00};
```

C++11 Structure Initialization :

Here = is optional.

personType person1 {“A R Danish”, 20, 45000.00};

Empty braces results in the individual members being set to 0.

personType person1 {};

Finally, narrowing is not allowed.

Can a structure use a string class member ?

The answer is yes unless you are using an obsolete compiler that does not support initialization of structures with string class members.

You can declare a structure like this :

```
#include <string>
struct peopleType
{
        std::string name;
        int age;
        float income;
};
```

Make sure the structure function has access to the std namespace.

Other Structure Properties :

C++ makes user-defined as similar as possible to built-in types.

For example,

You can pass structures as arguments to a function, you can have a function use a structure as a return value. You can assign one structure to another of the same type (Doing so causes each member of one structure to be set to the value of the corresponding member in the other structure, even if the member is an array. This kind of assignment is called memberwise assignment.).

See the program – StructTwo.cpp

You can combine the definition of a structure form with the creation of structure variables.

```
struct parking
{
        Int keyNumber;
        char carName[20];
} john_clark, mac_josh; // two parking spot variable
```

You even can initialize a variable you create in this fashion.

```
struct parking
{
        Int keyNumber;
        char carName[20];
}john_clark =
{
        1,
        Ibiza
};
```

However, keeping the structure definition separate from the variable declarations usually makes a program easier to read and follow.

You can create structures with no type name :

```
struct  // no tag
{
        int x;
        int y;
} coOrdinates; // a structure variable
```

This creates one structure variable called position. You can access its members with the membership operator, as in position.x, but there is no general name for the type.

Array Of Structures :

```
personType persons[100];  // array of 100 personType structures

cout << persons[1].name; // you can access like this.
```

Or

```
personType persons[2] =
{
        {"A R Danish", 19, 45000},
        {"Nerd Guy", 22, 85000}
};
```

See the programs – StructThree.cpp

Bit Field In Structures:

C++,like C, enables you to specify structure members that occupy a particular number of bits. This can be handy for creating a data structure that corresponds, say, to a register on some hardware device.

The field type should be an integral or enumeration type, and a colon followed by a number indicates the actual number of bits to be used. You can use unnamed fields to provide spacing. Each member is termed a **bit field**.

For example :

```
struct bit_field
{
        unsigned int num : 4; // 4 bits for ISBN value
        unsigned int : 4; // 4 bits unused
        bool switch : 1; // valid input (1 bit)
        bool is_driving : 1; // successful valid input
};
```

And initialize like this :

```
bit_field b1 = {45, false, true};
```

For accessing bit fields :

```
b1.is_driving;
```

Bit fields are typically used in low-level programming.

Unions :

A union is a data format that can hold different data types but only one type at a time. That is, whereas a structure can hold, say, an int and a long and a double, a union can hold an int or a long or a double.

The syntax is like that for a structure, but the meaning is different.

e.g.

```
union num_union

{

        int num1;

        float num2;

        double num3;

};
```

You can use union like this :

```
num_union union1;

union1.num1 = 15; // store an int

cout << union1.num1;

union1.num2 = 45.85; // store a double but int value is lost

cout << union1.num2;
```

So, union1 can serve as an int variable at a time and float variable at another time.

Size of the union is the size of the largest member.

One use for a union is to save space when a data item can use two or more formats but never simultaneously.

An anonymous union has no name; in essence, its members become variables that share the same address. Naturally, only one member can be current at a time.

```
struct widget
{
        char brand[20];
        int type;
        union               // anonymous union
        {
                long id_num;      // type 1 widgets
                char id_char[20]; // other widgets
        };
};
widget prize;
```

You can access like this :

```
prize.id_num;
prize.id_char;
```

Unions often (but not exclusively) are used to save memory space. That may not seem that necessary in these days of gigabytes of RAM and terabytes of storage, but not all C++ programs are written for such systems. C++ also is used for embedded systems, such as the processors used to control a toaster oven, an MP3 player, or a Mars rover. In these applications space may be at a premium.

Also unions often are used when working with operating systems or hardware data structures.

Enumeration:

The C++ enum facility provides an alternative to const for creating symbolic constants. It also lets you define new types but in a fairly restricted fashion. The syntax for enum resembles structure syntax.

e.g.

enum spectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet };

This statement does two things:

- It makes spectrum the name of a new type; spectrum is termed an enumeration, much as a struct variable is called a structure.
- It establishes red, orange, yellow, and so on, as symbolic constants for the integer values 0–7.These constants are called enumerators.

By default, enumerators are assigned integer values starting with 0 for the first enumerator,1 for the second enumerator, and so forth. You can override the default by explicitly assigning integer values.

The only valid values that you can assign to an enumeration variable without a type cast are the enumerator values used in defining the type.

band = blue; // valid, blue is an enumerator

band = 2000; // Invalid, 2000 not an enumerator

Only the assignment operator is defined for enumerations.

band = orange; //valid

++band; // Invalid

band = orange + red; // Invalid

Enumerators are of integer type and can be promoted to type int, but int types are not converted automatically to the enumeration type.

int color = blue; // valid, spectrum type promoted to int

band = 3; // Invalid, int not converted to spectrum

color = 3 + red; // Valid, red converted to int

Assigning 3 to band is a type error. But assigning green to band is fine because they are both type spectrum.

Again, some implementations do not enforce this restriction.

You can assign an int value to an enum, provided that the value is valid and that you use an explicit type cast.

band = spectrum(3); // typecast 3 to typecast spectrum

If you try to typecast an inappropriate value, the result will be undefined. It'll produce an error.

band = spectrum(1524); // undefined

In practice, enumerations are used more often as a way of defining related symbolic constants than as a means of defining new types.

If you plan to use just the constants and not create variables of the enumeration type, you can omit an enumeration type name.

e.g.

enum {red, orange, …., ultravoilet};

Setting Enumerator Value:

You can set enumerator values explicitly by using the assignment operator.

enum bits {one=1, two=2, four=4, eight=8};

The assigned value must be integers. You can also define only some of the enumerators explicitly.

enum number {first, second=10, third, fourth=1000};

In this case first is 0 by default. Subsequent uninitialized enumerators are larger by one than their predecessors. So, third would have the value 11 (10+1).

Another example like this:

enum numbers {zero, null=0, one, number_one=1};

Here, both zero and null are 0, and both one and number_one is 1.

In earlier versions of C++, you could only assign only int values (or values that promoted to int) to enumerators, but that restriction has been removed so that you use type *long* or even *long long* values.

Valid Range For Enumerations:

Originally, the only valid values for an enumeration were those named in the declaration. However, C++ has expanded the list of valid values that can be assigned to an enumeration variable through the use of a type cast. Each enumeration has a range, and you can assign any integer value in the range, even if it’s not an enumerator value, by using a type cast to an enumeration variable.

enum bits {one=1, two=2, four = 4, eight=8};

bits myflag;

myflag = bits(6); // valid, because 6 is in bit range

In these cases the range is between upper limit and lower limit.

To find the upper limit ($2^{\text{minimum greater value}}$ - 1). Lower limit 0 if given otherwise find using this technique.

For 115 upper limit is 127 (2^7 - 1).

For -6 lower limit is -7 (-2^3 + 1).

Pointers:

Pointers are variables that store addresses of values rather than the values themselves.

We just use & operator to a variable to get its location.

e.g.

If var is a variable then &var is its address.

See the programs – PointersOne.cpp

The particular implementation of cout shown here uses hexadecimal notation when displaying address values because that is the usual notation used to specify a memory address. (Some implementations use base 10 notation instead.)

The difference between two address is 4. It makes sense because pizza is type int, which uses 4 bytes. Different systems, of course, will give you different values for the address.

Using ordinary variables, then, treats the value as a named quantity and the location as a derived quantity.

Pointers and the C++ Philosophy:

Object-oriented programming differs from traditional procedural programming in that OOP emphasizes making decisions during runtime instead of during compile time. Runtime means while a program is running, and compile time means when the compiler is putting a program together. A runtime decision is like, when on vacation, choosing what sights to see depending on the weather and your mood at the moment, whereas a compile-time decision is more like adhering to a preset schedule, regardless of the conditions. Runtime decisions provide the flexibility to adjust to current circumstances. For example, consider allocating memory for an array. The traditional way is to declare an array. To declare an array in C++, you have to commit yourself to a particular array size. Thus, the array size is set when the program is compiled; it is a compile-time decision. Perhaps you think an array of 20 elements is sufficient 80% of the time but that occasionally the program will need to handle 200 elements. To be safe, you use an array with 200 elements. This results in your program wasting memory most of the time it’s used. OOP tries to make a program more flexible by delaying such decisions until runtime. That way, after the program is running, you can tell it you need only 20 elements one time or that you need 205 elements another time. In short, with OOP you would like to make the array size a runtime decision. To make this approach possible, the language has to allow you to create an array—or the equivalent— while the program runs. The C++ method, as you soon see, involves using the keyword new to request the correct amount of memory and using pointers to keep track of where the newly allocated memory is found. Making runtime decisions is not unique to OOP. But C++ makes writing the code a bit more straightforward than does C.

See the programs – PointerTwo.cpp

As you can see, the int variable value and the pointer variable p_value are just two sides of the same coin. The value variable represents the value as primary and uses the & operator to get the address, whereas the p_value variable represents the address as primary and uses the * operator to get the value.

int num = 45;

int *p_num = &num;

These are same.

num = *p_num = 23;

&num = p_num = 0X3ad6;

Incidentally, the use of spaces around the * operator are optional.

int *ptr;

or int* ptr;

or even int*ptr;

but here,

int* p1, p2;

p2 is an ordinary int not pointer.

You need an * for each pointer variable name.

In C++, the combination int * is a compound type, pointer-to-int.

See the programs – PointersThree.cpp

Danger awaits those who incautiously use pointers. One extremely important point is that when you create a pointer in C++,the computer allocates memory to hold an address, but it does not allocate memory to hold the data to which the address points. Creating space for the data involves a separate step. Omitting that step, as in the following, is an invitation to disaster.

```
long * fellow;         // create a pointer-to-long
*fellow = 223323;      // place a value in never-never land
```

Sure, fellow is a pointer. But where does it point? The code failed to assign an address to fellow. So where is the value 223323 placed? We can’t say. Because fellow wasn’t initialized, it could have any value. Whatever that value is, the program interprets it as the address at which to store 223323.If fellow happens to have the value 1200,then the computer attempts to place the data at address 1200,even if that happens to be an address in the middle of your program code. Chances are that wherever fellow points, that is not where you want to put the number 223323.This kind of error can produce some of the most insidious and hard-to-trace bugs.

Pointer Golden Rule: **Always initialize a pointer to a definite and appropriate address before you apply the dereferencing operator (*) to it.**

Pointers and Numbers:

Pointers are not integer types, even though computers typically handle addresses as integers.

Doing arithmetic operations on locations doesn’t make any sense.

So, you can’t simply assign an integer to a pointers:

```
int * pt;
```

```
pt = 0XB8000000; // type mismatch
```

C prior to C99 let you make statements like this. But C++ more stringently enforces type agreement.

you should use a type cast to convert the number to the appropriate address type.

```
pt = (int *) 0XB8000000; // now match
```

Allocating memory with *new*:

The true worth of pointers comes into play when you allocate unnamed memory during runtime to hold values. In this case, pointers become the only access to that memory. In C, you can allocate memory with the library function malloc().You can still do so in C++,but C++ also has a better way: the new operator.

```
int * pn = new int;
```

The new int part tells the program you want some new storage suitable for holding an int. The new operator uses the type to figure out how many bytes are needed. Then it finds the memory and returns the address.

int value;

int * pt = &value;

In both cases (pn and pt), you assign the address of an int to a pointer. In the second case, you can also access the int by name: value. In the first case, your only access is via the pointer.

The term “data object” is more general than the term “variable” because it means any block of memory allocated for a data item. Thus, a variable is also a data object, but the memory to which pn points is not a variable.

The general form for obtaining and assigning memory for a single data object, which can be a structure as well as a fundamental type, is this:

typeName * pointer_name = new typeName;

See the program –

Another point to note is that typically new uses a different block of memory than do the ordinary variable definitions that we have been using. Both the variables value and pd have their values stored in a memory region called the stack, whereas the memory allocated by new is in a region called the heap or free store.

Out Of memory

It’s possible that a computer might not have sufficient memory available to satisfy a new request. When that is the case, new normally responds by throwing an exception, an error handling technique. In older implementations, new returns the value 0. In C++, a pointer with the value 0 is called the null pointer. C++ guarantees that the null pointer never points to valid data, so it is often used to indicate failure for operators or functions that otherwise return usable pointers. C++ provides the tools to detect and respond to allocation failures.

Freeing Memory with *delete*:

Using new to request memory when you need it is just the more glamorous half of the C++ memory-management package. The other half is the delete operator, which enables you to return memory to the memory pool when you are finished with it. That is an important step toward making the most effective use of memory. Memory that you return, or free, can then be reused by other parts of the program.

```
int * ps = new int; // allocate memory with new

….. // use the memory

delete ps; // free memory with delete when done
```

You should always balance a use of new with a use of delete; otherwise, you can wind up with a memory leak— that is, memory that has been allocated but can no longer be used. If a memory leak grows too large, it can bring a program seeking more memory to a halt.

You should not attempt to free a block of memory that you have previously freed. The C++ Standard says the result of such an attempt is undefined, meaning that the consequences could be anything. Also you cannot use delete to free memory created by declaring ordinary variables.

e.g.

```
int * ps = new int; // ok

delete ps; // ok
delete ps; // not ok now

int val = 5; //ok

int * pi = &val; //ok

delete pi; // not allowed
```

You should use delete only to free memory allocated with new. However, it is safe to apply delete to a null pointer.

But don't do this too :

```
int * ps = new int; //allocate memory

int * pq = ps; // set second pointer to the same block

delete pq; // delete with second pointer
```

Ordinarily, you won't create two pointers to the same block of memory because that raises the possibility that you will mistakenly try to delete the same block twice. But ,using a second pointer does make sense when you work with a function that returns a pointer.

Using *new* to create Dynamic Arrays:

You can use new for larger data, such as arrays, strings and structures. This is where new is useful.

If you create an array by declaring it, the space is allocated when the program is compiled. Whether or not the program finally uses the array, the array is there, using up memory.

Allocating the array during compile time is called **static binding**, meaning that the array is built in to the program at compile time. But with new, you can create an array during **runtime** if you need it and skip creating the array if you don't need it. Or you can select an array size after the program is running. This is called **dynamic binding**, meaning that the array is created while the program is running. Such an array is called a **dynamic array**. With static binding, you must specify the array size when you write the program. With dynamic binding, the program can decide on an array size while the program runs.

Creating a Dynamic Array with *new*:

Creating dynamic array:

```
int * psome = new int [10]; // get a block of 10 ints
```

The new operator returns the address of the first element of the block. In this example, that value is assigned to the pointer psome.

Always try to balance the call to new with a call to delete when the program finishes using that block of memory.

The syntax for freeing the array using delete will be like this:

delete [] psome; //free a dynamic array

If you use new without brackets, you should use delete without brackets. If you use new with brackets, you should use delete with brackets.

In short :

- Don't use delete to free memory that new didn't allocate.
- Don't use delete to free the same block of memory twice in succession.
- Use delete [] if you used new [] to allocate an array.
- Use delete (no brackets) if you used new to allocate a single entity.
- It's safe to apply delete to the null pointer (nothing happens).

The general form for allocating and assigning memory for an array is this.

typeName * pointerName = new typeName [numElements];

Accessing a dynamic array:

The first element is no problem. Because psome points to the first element of the array, *psome is the value of the first element. That leaves nine more elements to access.

That is, you can use psome[0] instead of *psome for the first element, psome[1] for the second element, and so on.

You must be wondering why this method works?

The reason you is that C and C++ handle arrays internally by using pointers anyway. This near equivalence of arrays and pointers is one of the beauties of C and C++ (Sometimes it's also a problem but that's another story**).**

See the program – PointersFive.cpp

p3 = p3 + 1; // okay for pointers, wrong for array names

You can't change the value of an array name. But a pointer is a variable, hence you can change its value.

Subtracting one takes the pointer back to its original value so that the program can provide delete [] with the correct address.

Pointers, Arrays, and Pointer Arithmetic:

Adding one to an integer variable increases its value by one, but adding one to a pointer variable increases its value by the number of bytes of the type to which it points. Adding one to a pointer to double adds 8 to the numeric value on systems with 8-byte double, whereas adding one to a pointer-to-short adds two to the pointer value if short is 2 bytes.

See the Programs – PointersSixAdd.cpp

Adding one to a pointer variable increases its value by the number of bytes of the type to which it points.

wherever you use array notation, C++ makes the following conversion:

arrayname[i] becomes *(arrayname + i)

And if you use a pointer instead of an array name, C++ makes the same conversion:

pointername[i] becomes *(pointername + i)

Thus, in many respects you can use pointer names and array names in the same way. One difference is that you can change the value of a pointer, whereas an array name is a constant.

pointername = pointername + 1; // valid

arrayname = arrayname + 1;   // not allowed

A second difference is that applying the sizeof operator to an array name yields the size of the array, but applying sizeof to a pointer yields the size of the pointer, even if the pointer points to the array. As you saw in the program.

Address of the array:

```
short tell[10];        // tell an array of 20 bytes
cout << tell << endl;          // displays &tell[0]
cout << &tell << endl; // displays address of whole array
```

Numerically, these two addresses are the same, but conceptually &tell[0], and hence tell, is the address of a 2-byte block of memory, whereas &tell is the address of a 20byte block of memory.

So the expression tell + 1 adds 2 to the address value, whereas &tell + 1 adds 20 to the address value. Another way of expressing this is to say that tell is type pointer-to-short, or short *, and &tell is type pointer-to-array-of-20-shorts, or short (*)[20].

Now you might be wondering about the genesis of that last type description. First, here is how you could declare and initialize a pointer of that type:

short (*pas)[20] = &tell; // pas points to array of 20 shorts

Pointers and Strings:

The special relationship between arrays and pointers extends to C-style strings.

char flower[10] = "rose";

cout << flower << "s are red\n";

The crucial element here is not that flower is an array name but that flower acts as the address of a char. This implies that you can use a pointer-to-char variable as an argument to cout also because it, too, is the address of a char. Of course, that pointer should point to the beginning of a string.

With cout and with most C++ expressions, the name of an array of char, a pointer-to-char, and a quoted string constant are all interpreted as the address of the first character of a string.

See the program – PointersSevenString.cpp

When you read a string into a program-style string, you should always use the address of previously allocated memory. This address can be in the form of an array name or of a pointer that has been initialized using new.

Use strcpy() or strncpy(), not the assignment operator, to assign a string to an array.

Using *new* to Create Dynamic Structures:

Again, dynamic means the memory is allocated during runtime, not at compile time.

Using new with structures has two parts: creating the structure and accessing its members.

To create a structure, you use the structure type with new.

structName * ptr = new structName;

This assigns to ptr the address of a chunk of free memory large enough to hold a structure of the structName type.

When you create a dynamic structure, you can't use the dot membership operator with the structure name because the structure has no name. All you have is its address. C++ provides an operator for these situations: the arrow membership operator (->). This operator does for pointer to structures what the dot operator does for structure names.

Tip

Sometimes new C++ users become confused about when to use the dot operator and when to use the arrow operator to specify a structure member. The rule is simple: If the structure identifier is the name of a structure, use the dot operator. If the identifier is a pointer to the structure, use the arrow operator.

See the program – StructPtrOne.cpp

Suppose your program has to read 1,000 strings and that the largest string might be 79 characters long, but most of the strings are much shorter than that. If you used char arrays to hold the strings, you'd need 1,000 arrays of 80 characters each. That's 80,000 bytes, and much of that block of memory would wind up being unused. Alternatively, you could create an array of 1,000 pointers to char and then use new to allocate only the amount of memory needed for each string. That could save tens of thousands of bytes. Instead of having to use a large array for every string, you fit the memory to the input. Even better, you could also use new to find space to store only as many pointers as needed. Well, that's a little too ambitious for right now.

See the program – StructPtrDelte.cpp

Automatic Storage, Static Storage and Dynamic Storage:

C++ has three ways of managing memory for data, depending on the method used to allocate memory: automatic storage, static storage, and dynamic storage, sometimes called the **free store** or **heap**. Data objects allocated in these three ways differ from each other in how long they remain in existence.

C++ adds a fourth form called **thread storage** too.

Automatic Storage:

- Actually, automatic values are local to the block that contains them. A block is a section of code enclosed between braces.
- Automatic variables typically are stored on a stack. This means that when program execution enters a block of code, its variables are added consecutively to the stack in memory and then are freed in reverse order when execution leaves the block (LIFO process). So the stack grows and shrinks as execution proceeds.

Static Storage:

Static storage is storage that exists throughout the execution of an entire program.

There are two ways to make a variable static. One is to define it externally, outside a function. The other is to use the keyword static when declaring a variable.

The main point about automatic and static storage is that these methods rigidly define the lifetime of a variable. Either the variable exists for the entire duration of a program (a static variable) or it exists only while a particular function is being executed (an automatic variable).

Dynamic Storage:

The new and delete operators provide a more flexible approach than automatic and static variables. They manage a pool of memory which C++ refers to as the free store or heap. This pool is separate from the memory used for static and automatic variables.

The lifetime of the data is not tied arbitrarily to the life of the program or the life of a function. Using new and delete together gives you much more control over how a program uses memory than does using ordinary variables.

Stacks, Heaps and Memory Leaks:

If you don’t call delete with new it can create **memory leak**.

Even the best programmers and software companies create memory leaks. To avoid them, it’s best to get into the habit of joining your new and delete operators immediately, planning for and entering the deletion of your construct as soon as you dynamically allocate it on the free store. C++ smart pointers automate this task.

Note:

Pointers are among the most powerful of C++ tools. They are also the most dangerous because they permit computer-unfriendly actions, such as using an uninitialized pointer to access memory or attempting to free the same memory block twice. Furthermore, until you get used to pointer notation and pointer concepts through practice, pointers can be confusing. Because pointers are an important part of C++ programming, we'll discuss it again and again.

See the program – ArrStructPtr.cpp

For combination of arrays, structures and pointers.

Array Alternatives - The *vector* template class:

The vector template class is similar to the string class in that it is a dynamic array. You can set the size of a vector object during runtime, and you can append new data to the end or insert new data in the middle.

Basically, it's an alternative to using new to create a dynamic array. Actually, the vector class does use new and delete to manage memory, but it does so automatically.

1. To use a vector object, you need to include the vector header file.
2. The vector identifier is part of the std namespace, so you can use a using directive, a using declaration, or std::vector.

3. Templates use a different syntax to indicate the type of data stored.
4. The vector class uses a different syntax to indicate the number of elements.

e.g.

```
#include <vector>


using namespace std;

vector<int> vi; // create a zero-size array of int
```

In general, the following declaration creates a vector object vt that can hold n_elem elements of type typeName :

```
vector<typeName> vt(n_elem);
```

The parameter n_elem can be an integer constant or an integer variable.

The array Template Class (C++11):

```
array<int, 5> ai; // create array objects of 5 ints

array<double, 4> ad = {4.5, 7.8, 2.4, 5.6};
```

More general, the following declaration creates an array object arr with

n_elem elements of typeName :

```
array< typeName , n_elem > arr;
```

Unlike the case for vector, n_elem can’t be a variable.

With C++11, you can use list-initialization with vector and array objects. However, that was not an option with C++98 vector objects.

Comparing **Arrays,vector Objects, and Array Objects :**

See the program – YoursChoice.cpp

Just as you can use the getline() member function with the cin object, you can use the at() member function with objects of the vector or array type:

arr2.at(1) = 2.3; // assign 2.3 to arr2[1]

The difference between using bracket notation and the at() member function is that if you use at(),an invalid index is caught during runtime and the program, by default, aborts. This added checking does come at the cost of increased run time, which is why C++ gives you the option of using either notation. More than that, these classes offer ways of using objects that reduce the chances of accidental range errors.

THANK YOU