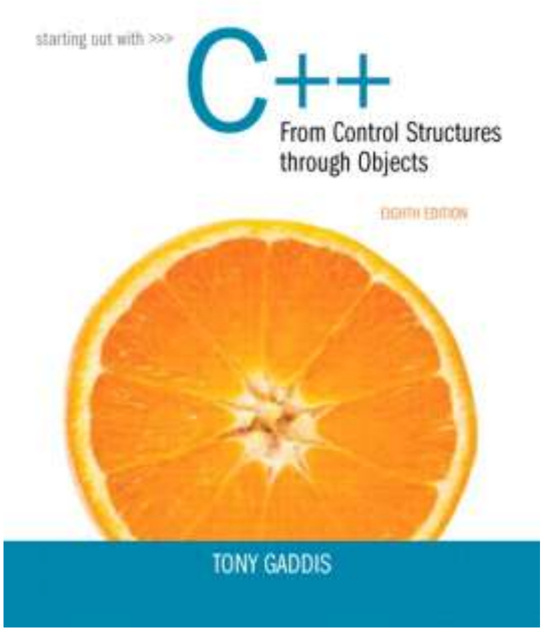
Chapter 9:

Pointers



9.1

Getting the Address of a Variable

Getting the Address of a Variable

Each variable in program is stored at a unique address
Use address operator & to get address of a variable:

```
int num = -99;
cout << &num;        // prints address
                     // in hexadecimal
```

Address-of operator (&)

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign (&), known as *address-of operator*. For example:

```
foo = &myvar;
```

This would assign the address of variable myvar to foo; by preceding the name of the variable myvar with the *address-of operator* (&), we are no longer assigning the content of the variable itself to foo, but its address.

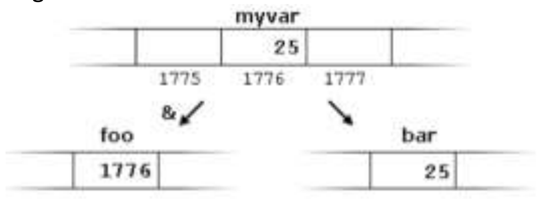
http://www.cplusplus.com

The actual address of a variable in memory cannot be known before runtime, but let's assume, in order to help clarify some concepts, that myvar is placed during runtime in the memory address 1776.

In this case, consider the following code fragment:

```
1. myvar = 25;
2. foo = &myvar;
3. bar = myvar;
```

The values contained in each variable after the execution of this are shown in the following diagram:



http://www.cplusplus.com

The variable that stores the address of another variable (like foo in the previous example) is what in C++ is called a *pointer*. Pointers are a very powerful feature of the language that has many uses in lower level programming.

http://www.cplusplus.com

Program 9-1

```
// This program uses the & operator to determine a variable's
// address and the sizeof operator to determine its size.
#include <iostream>
using namespace std;

int main()
{
   int x = 25;

   cout << "The address of x is " << &x << endl;
   cout << "The size of x is " << sizeof(x) << " bytes\n";
   cout << "The value in x is " << x << endl;
   return 0;
}
```

```
The address of x is 0x8f05
The size of x is 4 bytes
The value in x is 25
```

9.2

Pointer Variables

Pointer Variables

Pointer variable : Often just called a pointer, it's a variable that holds an address

Because a pointer variable holds the address of another piece of data, it "points" to the data

Declaring Pointers

Pointers must be declared before they can be used, just like a normal variable. The syntax of declaring a pointer is to place a * in front of the name. A pointer is associated with a type (such as int and double) too.

```
type *ptr;           // Declare a pointer variable called ptr as a pointer of type
                     // or
 type* ptr;
                     // or
type * ptr;
```

For example,

```
int * iPtr;          // Declare a pointer variable  iPtr pointing to an int
double * dPtr;       // Declare a pointer to a double
```

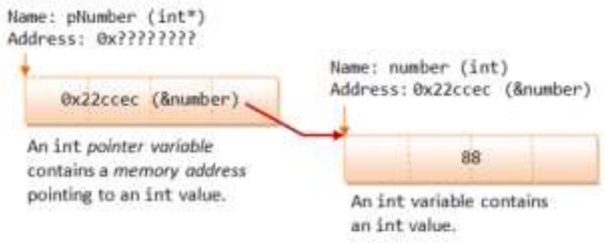
Initializing Pointers via the Address-Of Operator (&)

When you declare a pointer variable, its content is not initialized. In other words, it contains an address of "somewhere", which is of course not a valid location.

This is dangerous! You need to initialize a pointer by assigning it a valid address. This is normally done via the *address-of operator* (&).

ntu.edu.sg

```
int number = 88;          // An int variable with a value

int * pNumber;            // Declare a pointer variable called pNumber
                          //pointing to an int (or int pointer)

 pNumber = &number;       // Assign the address of the variable number to
                          //pointer pNumber
```


Name: pNumber (int*)
Address: 0x????????
0x22ccec (&number)
An int pointer variable contains a memory address pointing to an int value.
Name: number (int)
Address: 0x22ccec (&number)
88
An int variable contains an int value.

ntu.edu.sg

Dereference operator (*)

A *pointer* stores the address of another variable. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the *dereference operator* (*).

The operator itself can be read as "value pointed to by".

http://www.cplusplus.com

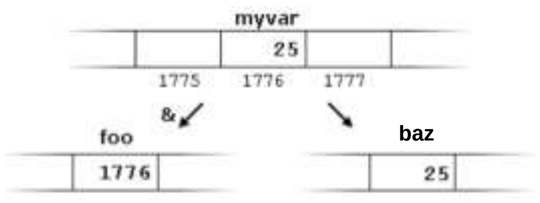
Dereference operator (*)

For example,

```
myvar = 25;
foo = &myvar;
```

the following statement:

```
baz = *foo;
```

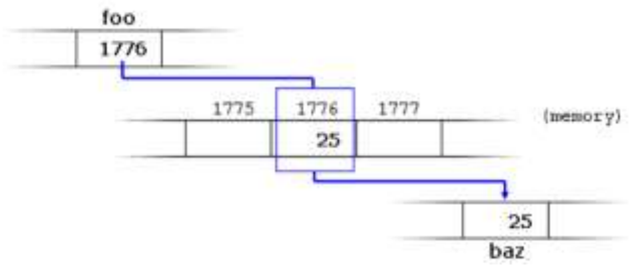


This could be read as: "baz equal to value pointed to by foo", or " baz is equal to what foo is pointing to ".

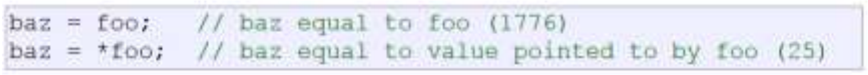
The statement would assign the value 25 to baz, since foo is 1776, and the value pointed to by 1776 would be 25.

http://www.cplusplus.com

```
myvar = 25;
foo = &myvar;
baz = *foo;
```



It is important to clearly differentiate that foo refers to the value 1776, while *foo (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the *dereference operator*



```
baz = foo;   // baz equal to foo (1776)
baz = *foo;  // baz equal to value pointed to by foo (25)
```

http://www.cplusplus.com

The reference and dereference operators are thus complementary:

- & is the *address-of operator*, and can be read simply as "address of"
- * is the *dereference operator*, and can be read as "value pointed to by“

Earlier, we performed the following two assignment operations:

1. myvar = 25;
2. foo = &myvar;

Right after these two statements, all of the following expressions would give true as result:

1. myvar == 25
2. &myvar == 1776
3. foo == 1776
4. *foo == 25 (the value foo is pointing to)
5. *foo == myvar

http://www.cplusplus.com

The Indirection Operator

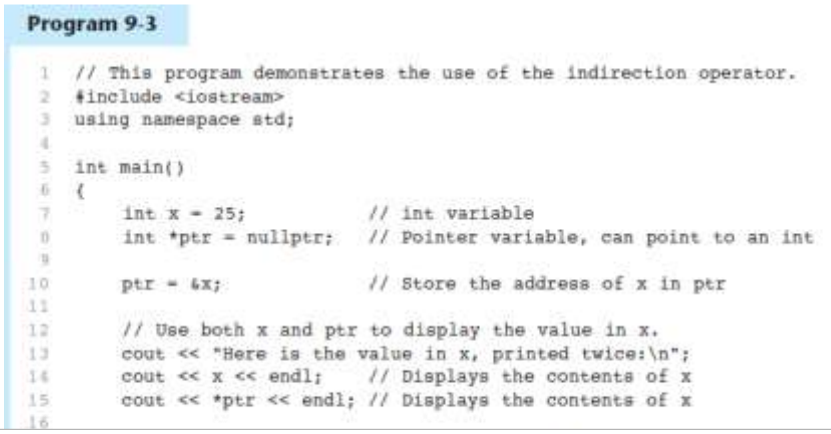
The indirection operator (`*`) dereferences a pointer.
It allows you to access the item that the pointer points to.

```
int x = 25;
int *intptr = &x;
cout << *intptr << endl;
```

This prints 25.

"value pointed to by intptr "

The Indirection Operator in Program 9-3

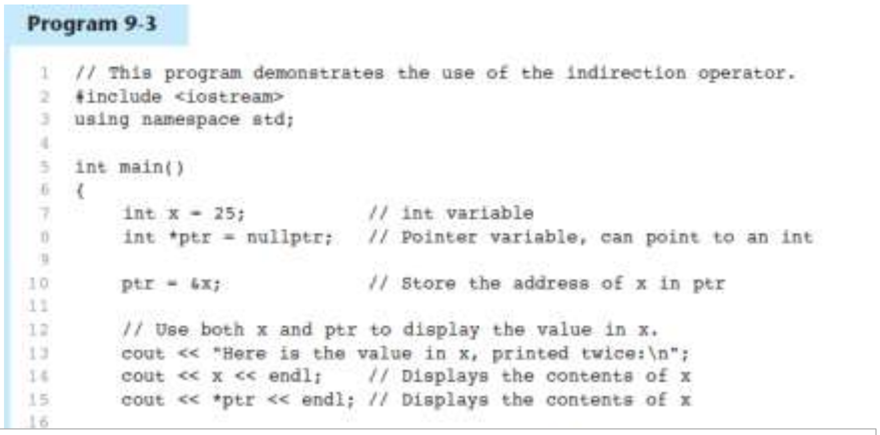


Program 9-3

```
 1  // This program demonstrates the use of the indirection operator.
 2  #include <iostream>
 3  using namespace std;
 4
 5  int main()
 6  {
 7      int x = 25;            // int variable
 8      int *ptr = nullptr;    // Pointer variable, can point to an int
 9
10      ptr = &x;              // Store the address of x in ptr
11
12      // Use both x and ptr to display the value in x.
13      cout << "Here is the value in x, printed twice:\n";
14      cout << x << endl;     // Displays the contents of x
15      cout << *ptr << endl; // Displays the contents of x
16
```

Take a closer look at the statement in line 10:
ptr = &x;
This statement assigns the address of the x variable to the ptr variable.

The Indirection Operator in Program 9-3



Program 9-3

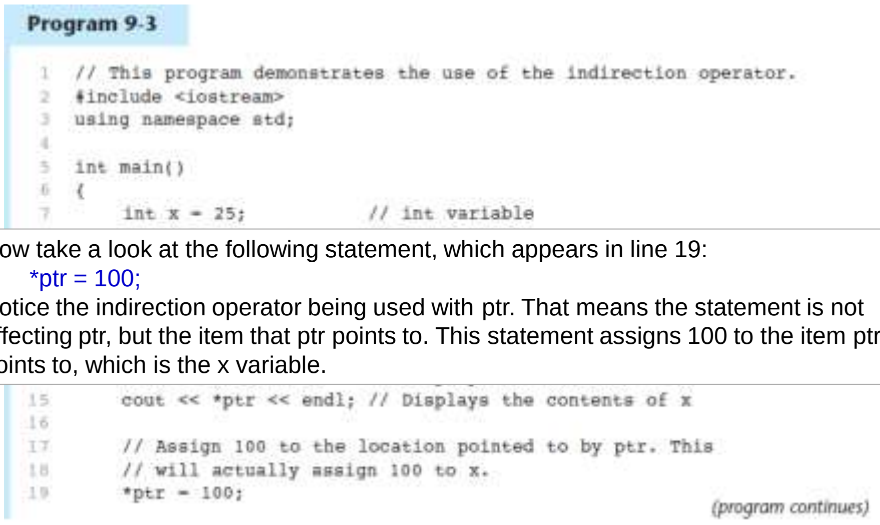
```
1   // This program demonstrates the use of the indirection operator.
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7       int x = 25;            // int variable
8       int *ptr = nullptr;    // Pointer variable, can point to an int
9
10      ptr = &x;              // Store the address of x in ptr
11
12      // Use both x and ptr to display the value in x.
13      cout << "Here is the value in x, printed twice:\n";
14      cout << x << endl;     // Displays the contents of x
15      cout << *ptr << endl; // Displays the contents of x
16
```

Now look at line 15:

cout << *ptr << endl; // Displays the contents of x

When you apply the indirection operator (*) to a pointer variable, you are working, not with the pointer variable itself, but with the item it points to.

The Indirection Operator in Program 9-3



Program 9-3

```
1   // This program demonstrates the use of the indirection operator.
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7       int x = 25;            // int variable
```

Now take a look at the following statement, which appears in line 19:

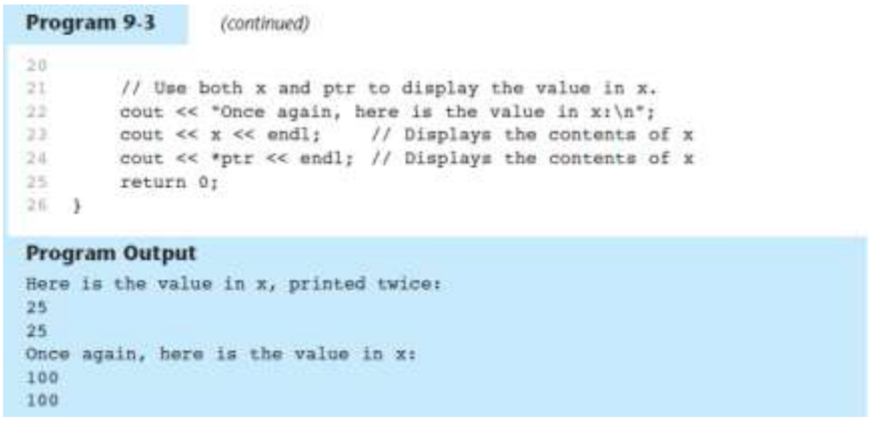
*ptr = 100;

Notice the indirection operator being used with ptr. That means the statement is not affecting ptr, but the item that ptr points to. This statement assigns 100 to the item ptr points to, which is the x variable.

```
15      cout << *ptr << endl; // Displays the contents of x
16
17      // Assign 100 to the location pointed to by ptr. This
18      // will actually assign 100 to x.
19      *ptr = 100;
```

(program continues)

The Indirection Operator in Program 9-3

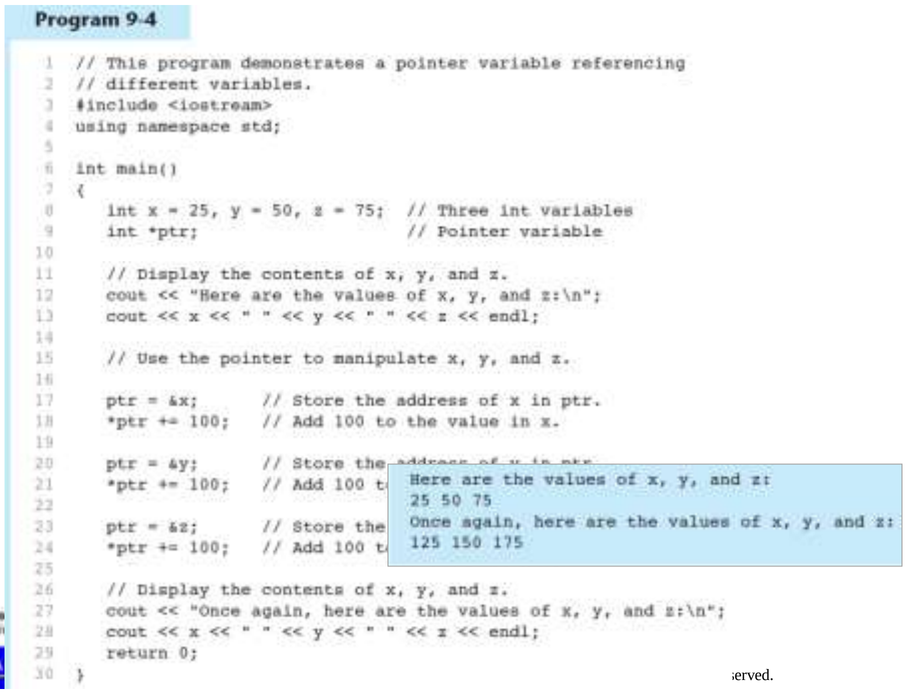


Program 9-3 *(continued)*

```
    // Use both x and ptr to display the value in x.
    cout << "Once again, here is the value in x:\n";
    cout << x << endl;     // Displays the contents of x
    cout << *ptr << endl;  // Displays the contents of x
    return 0;
}
```

Program Output

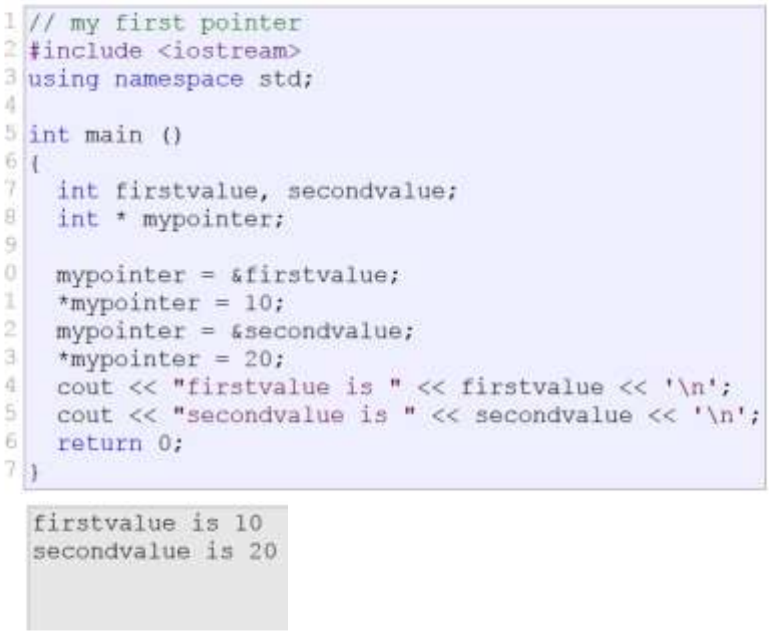
```
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
```



Program 9-4

```
// This program demonstrates a pointer variable referencing
// different variables.
#include <iostream>
using namespace std;

int main()
{
   int x = 25, y = 50, z = 75;  // Three int variables
   int *ptr;                    // Pointer variable

   // Display the contents of x, y, and z.
   cout << "Here are the values of x, y, and z:\n";
   cout << x << " " << y << " " << z << endl;

   // Use the pointer to manipulate x, y, and z.

   ptr = &x;       // Store the address of x in ptr.
   *ptr += 100;    // Add 100 to the value in x.

   ptr = &y;       // Store the address of y in ptr.
   *ptr += 100;    // Add 100 t

   ptr = &z;       // Store the
   *ptr += 100;    // Add 100 t

   // Display the contents of x, y, and z.
   cout << "Once again, here are the values of x, y, and z:\n";
   cout << x << " " << y << " " << z << endl;
   return 0;
}
```

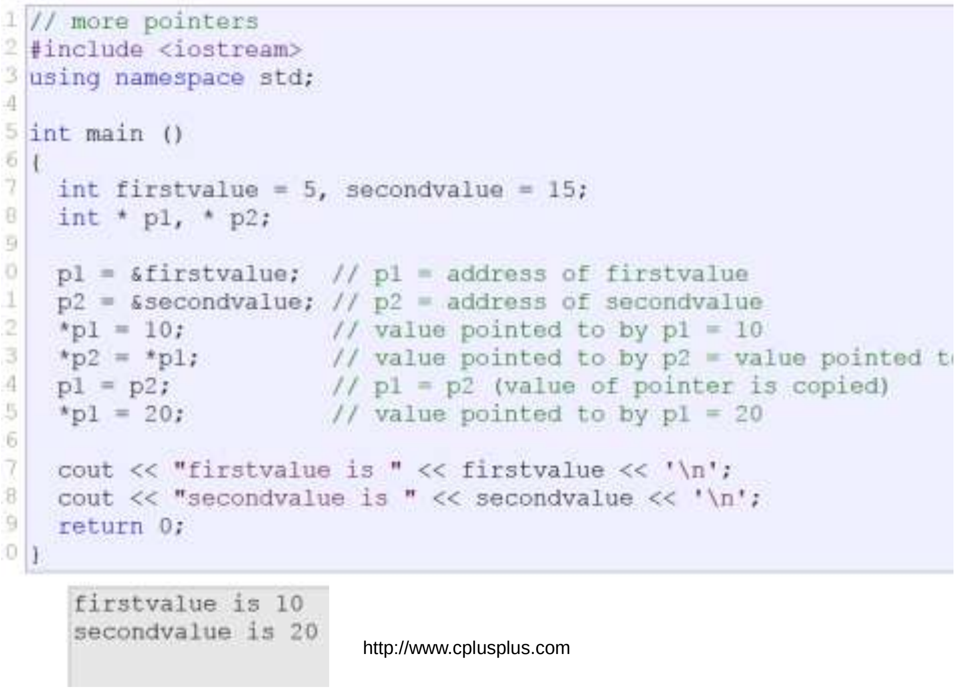
```
Here are the values of x, y, and z:
25 50 75
Once again, here are the values of x, y, and z:
125 150 175
```



```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

http://www.cplusplus.com



```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 = value pointed t
  p1 = p2;           // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

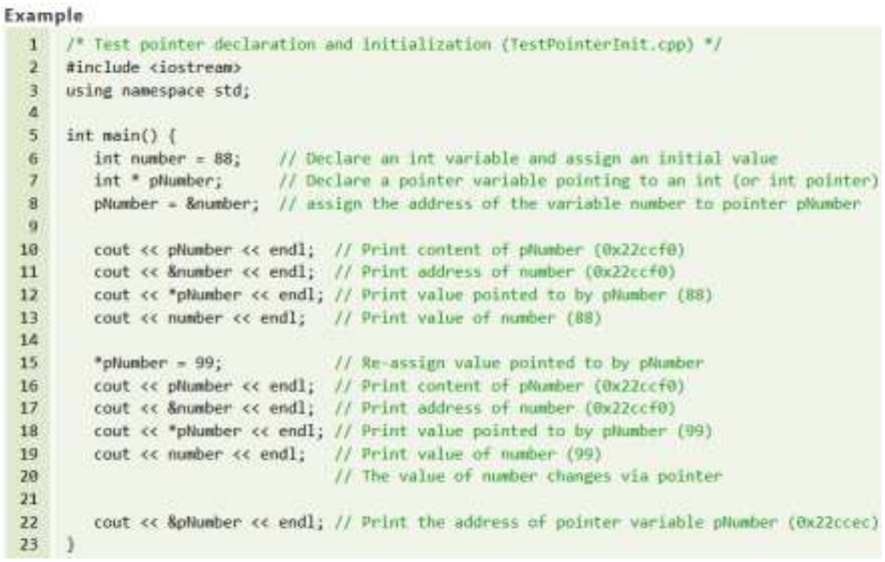
http://www.cplusplus.com

Pointer has a Type Too

A pointer is associated with a type (of the value it points to), which is specified during declaration. A pointer can only hold an address of the declared type; it cannot hold an address of a different type.

```
int i = 88;
double d = 55.66;
int * iPtr = &i;         // int pointer pointing to an int value
double * dPtr = &d; // double pointer pointing to a double value

iPtr = &d;           // ERROR, cannot hold address of different type
dPtr = &i;           // ERROR
iPtr = i;            // ERROR, pointer holds address of an int, NOT int value

int j = 99;
iPtr = &j;           // You can change the address stored in a pointer
```

http://www.ntu.edu.sg



Example

```
/* Test pointer declaration and initialization (TestPointerInit.cpp) */
#include <iostream>
using namespace std;

int main() {
   int number = 88;    // Declare an int variable and assign an initial value
   int * pNumber;      // Declare a pointer variable pointing to an int (or int pointer)
   pNumber = &number;  // assign the address of the variable number to pointer pNumber

   cout << pNumber << endl;  // Print content of pNumber (0x22ccf0)
   cout << &number << endl;  // Print address of number (0x22ccf0)
   cout << *pNumber << endl; // Print value pointed to by pNumber (88)
   cout << number << endl;   // Print value of number (88)

   *pNumber = 99;            // Re-assign value pointed to by pNumber
   cout << pNumber << endl;  // Print content of pNumber (0x22ccf0)
   cout << &number << endl;  // Print address of number (0x22ccf0)
   cout << *pNumber << endl; // Print value pointed to by pNumber (99)
   cout << number << endl;   // Print value of number (99)
                             // The value of number changes via pointer

   cout << &pNumber << endl; // Print the address of pointer variable pNumber (0x22ccec)
}
```

http://www.ntu.edu.sg

Pointer initialization

Pointers can be initialized to point to specific locations at the very moment they are defined:

```
int myvar;
int * myptr = &myvar;
```

The resulting state of variables after this code is the same as after:

```
int myvar;
int * myptr;
myptr = &myvar;
```

When pointers are initialized, what is initialized is the address they point to (i.e., myptr), never the value being pointed (i.e., *myptr). Therefore, the code above shall not be confused with:

```
int myvar;
int * myptr;
*myptr = &myvar;
```

(invalid code)

http://www.cplusplus.com

Pointers can be initialized either to the address of a variable (such as in the case above), or to the value of another pointer (or array):

```
int myvar;
int *foo = &myvar;
int *bar = foo;
```

http://www.cplusplus.com

Initializing Pointers

Can initialize at definition time:

```
int num, *numptr = &num;
int val[3], *valptr = val;
```

Cannot mix data types:

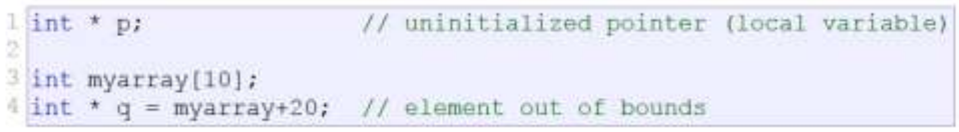
```
double cost;
int *ptr = &cost; // won't work
```

Can test for an invalid address for `ptr` with:

```
if (!ptr) ...
```

Invalid pointers

In principle, pointers are meant to point to valid addresses, such as the address of a variable or the address of an element in an array. But pointers can actually point to any address, including addresses that do not refer to any valid element. Typical examples of this are *uninitialized pointers* and pointers to nonexistent elements

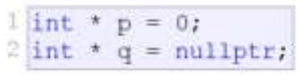


```
int * p;                // uninitialized pointer (local variable)

int myarray[10];
int * q = myarray+20;   // element out of bounds
```

http://www.cplusplus.com

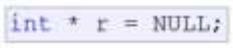
Null pointers

But, sometimes, a pointer really needs to explicitly point to nowhere, and not just an invalid address. For such cases, there exists a special value that any pointer type can take: the *null pointer value*. This value can be expressed in C++ in two ways: either with an integer value of zero, or with the nullptr keyword:



```
int * p = 0;
int * q = nullptr;
```

Here, both p and q are *null pointers*, meaning that they explicitly point to nowhere, and they both actually compare equal. It is also quite usual to see the defined constant NULL be used in older code to refer to the *null pointer* value:



```
int * r = NULL;
```

http://www.cplusplus.com

Null Pointers

You can initialize a pointer to 0 or nullptr or NULL, i.e., it points to nothing. It is called a *null pointer*. Dereferencing a null pointer (*p) causes an STATUS_ACCESS_VIOLATION exception.

```
int * iPtr = 0;           // Declare an int pointer, and initialize the pointer
                          // to point to nothing
cout << *iPtr << endl;    // ERROR! STATUS_ACCESS_VIOLATION exception

int * p = nullptr;
int * p = NULL;           // Also declare a NULL pointer points to nothing
```

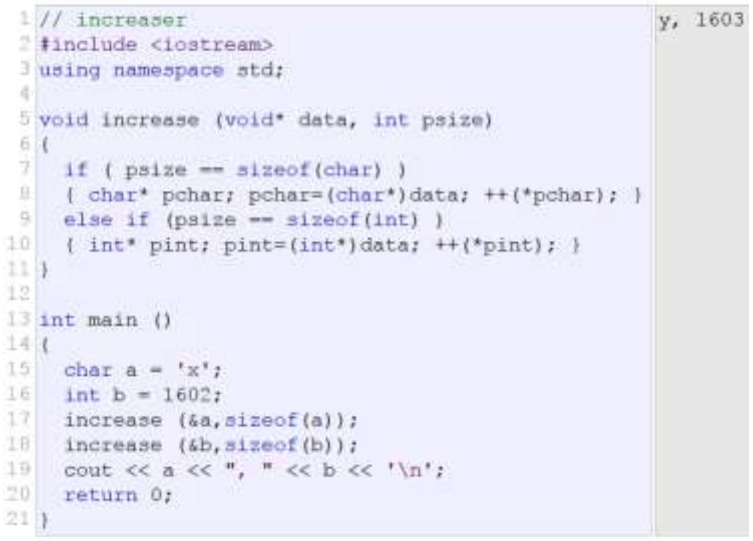
Initialize a pointer to null during declaration is a good software engineering practice.

Void pointers

The void type of pointer is a special type of pointer. In C++, void represents the absence of type. Therefore, void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).

For that reason, any address in a void pointer needs to be transformed into some other pointer type that points to a concrete data type before being dereferenced.

http://www.cplusplus.com



```
// increaser
#include <iostream>
using namespace std;

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); }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << '\n';
  return 0;
}
```

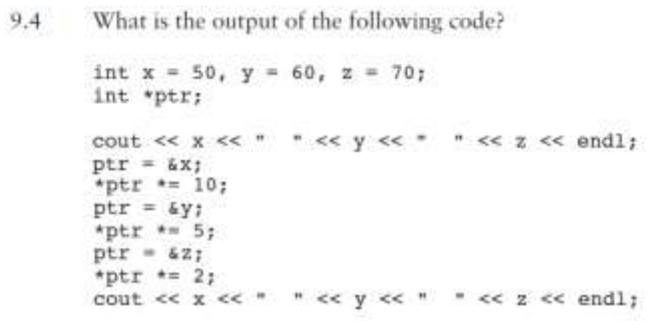
```
y, 1603
```

http://www.cplusplus.com

Do not confuse *null pointers* with void pointers!

A *null pointer* is a value that any pointer can take to represent that it is pointing to "nowhere“.

A void pointer is a type of pointer that can point to somewhere without a specific type.



9.4 What is the output of the following code?

```
int x = 50, y = 60, z = 70;
int *ptr;

cout << x << "   " << y << "   " << z << endl;
ptr = &x;
*ptr *= 10;
ptr = &y;
*ptr *= 5;
ptr = &z;
*ptr *= 2;
cout << x << "   " << y << "   " << z << endl;
```

500 300 140

9.7 Assume `pint` is a pointer variable. Is each of the following statements valid or invalid? If any is invalid, why?

```
A) pint++;
B) --pint;
C) pint /= 2;                X
D) pint *= 4;                X
E) pint += x;  // Assume x is an int.
```

9.8 Is each of the following definitions valid or invalid? If any is invalid, why?

```
A) int ivar;
   int *iptr = &ivar;
B) int ivar, *iptr = &ivar;  X
C) float fvar;
   int *iptr = &fvar;        X
D) int nums[50], *iptr = nums;
E) int *iptr = &ivar;        X
   int ivar;
```

9.3

The Relationship Between Arrays and Pointers

Pointers and arrays

The concept of arrays is related to that of pointers. For example, consider these two declarations:

```
int myarray [20];
int * mypointer;
```

The following assignment operation would be valid:

```
mypointer = myarray;
```

After that, mypointer and myarray would be equivalent and would have very similar properties. The main difference being that mypointer can be assigned a different address, whereas myarray can never be assigned anything, and will always represent the same block of 20 elements of type int. Therefore, the following assignment would not be valid:

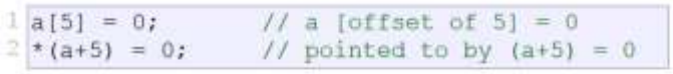
```
myarray = mypointer;
```

An example that mixes arrays and pointers:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

In the chapter about arrays, brackets ([]) were explained as specifying the index of an element of the array. Well, in fact these brackets are a dereferencing operator known as *offset operator*. They dereference the variable they follow just as * does, but they also add the number between brackets to the address being dereferenced. For example:



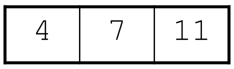
```
a[5] = 0;          // a [offset of 5] = 0
*(a+5) = 0;        // pointed to by (a+5) = 0
```

These two expressions are equivalent and valid, not only if a is a pointer, but also if a is an array. Remember that if an array, its name can be used just like a pointer to its first element.

The Relationship Between Arrays and Pointers

Array name is starting address of array

```
int vals[] = {4, 7, 11};
```



```
cout << vals;           // displays
```
starting address of `vals`: `0x4a00` `// 0x4a00`
```
cout << vals[0];        // displays 4
```

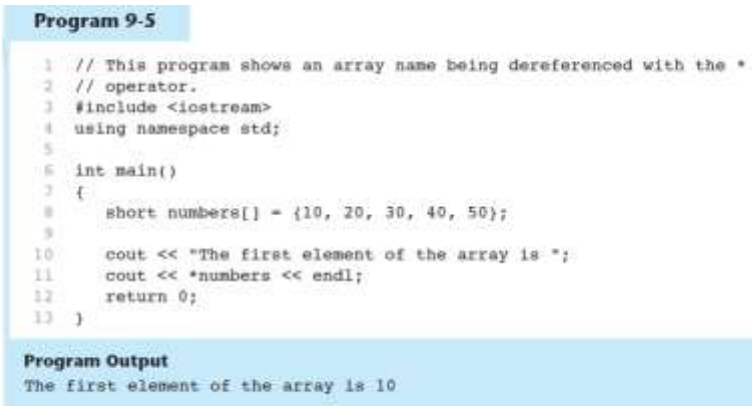
The Relationship Between Arrays and Pointers

Array name can be used as a pointer constant:

```
int vals[] = {4, 7, 11};
cout << *vals;     // displays 4
```

Pointer can be used as an array name:

```
int *valptr = vals;
cout << valptr[1]; // displays 7
```

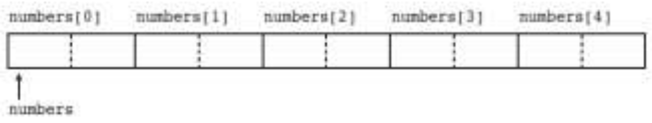


Program 9-5

```
// This program shows an array name being dereferenced with the *
// operator.
#include <iostream>
using namespace std;

int main()
{
   short numbers[] = {10, 20, 30, 40, 50};

   cout << "The first element of the array is ";
   cout << *numbers << endl;
   return 0;
}
```

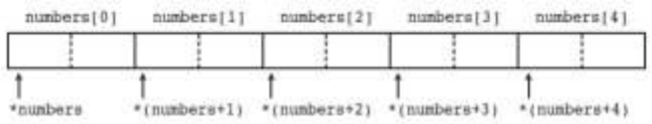
Program Output

```
The first element of the array is 10
```



If you add two to numbers, the result is numbers + 2 * sizeof(short), and so forth. On a PC, this means the following are true, because short integers typically use two bytes:

```
*(numbers + 1) is actually *(numbers + 1 * 2)
*(numbers + 2) is actually *(numbers + 2 * 2)
*(numbers + 3) is actually *(numbers + 3 * 2)
```



The only difference between array names and pointer variables is that you cannot change the address an array name points to. For example, consider the following definitions:

```
double readings[20], totals[20];
double *dptr;
```

These statements are legal:

```
dptr = readings;     // Make dptr point to readings.
dptr = totals;       // Make dptr point to totals.
```

But these are illegal:

```
readings = totals;   // ILLEGAL! Cannot change readings.
totals = dptr;       // ILLEGAL! Cannot change totals.
```

Array names are *pointer constants.* You can't make them point to anything but the array they represent.

9.5 Rewrite the following loop so it uses pointer notation (with the indirection operator) instead of subscript notation.

```
for (int x = 0; x < 100; x++)
   cout << arr[x] << endl;
```

```
*(arr + x)
```

9.4

Pointer Arithmetic

Types have different sizes. Let's imagine that in a given system, char takes 1 byte, short takes 2 bytes, and long takes 4.

Suppose now that we define three pointers in this compiler:

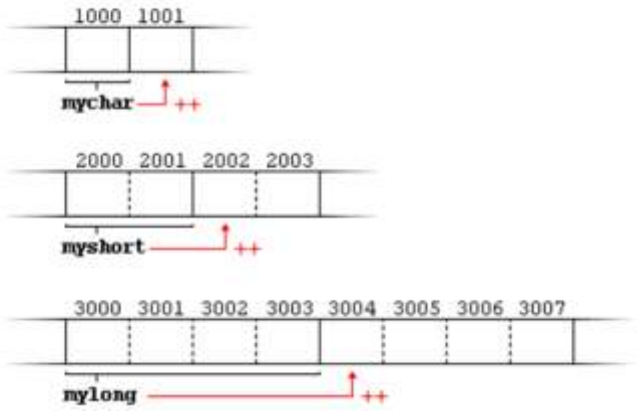
```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to the memory locations 1000, 2000, and 3000, respectively.

Therefore, if we write:

```
++mychar;
++myshort;
++mylong;
```

mychar, would contain the value 1001. myshort would contain the value 2002, and mylong would contain 3004, even though they have each been incremented only once. The reason is that, when adding one to a pointer, the pointer is made to point to the following element of the same type.



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we wrote:

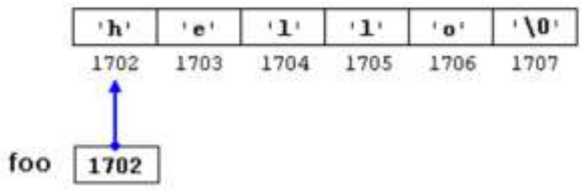
```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

Pointers and string literals

String literals are arrays of the proper array type to contain all its characters plus the terminating null-character, with each of the elements being of type const char (as literals, they can never be modified). For example:

```
const char * foo = "hello";
```

This declares an array with the literal representation for "hello", and then a pointer to its first element is assigned to foo. If we imagine that "hello" is stored at the memory locations that start at address 1702, we can represent the previous declaration as:



Note that here foo is a pointer and contains the value 1702, and not 'h', nor "hello", although 1702 indeed is the address of both of these.

```
const char * foo = "hello";
```

The pointer foo points to a sequence of characters. And because pointers and arrays behave essentially in the same way in expressions, foo can be used to access the characters in the same way arrays of null-terminated character sequences are. For example:

```
*(foo+4)
foo[4]
```

Both expressions have a value of 'o' (the fifth element of the array).

9.6

Comparing Pointers

Comparing Pointers

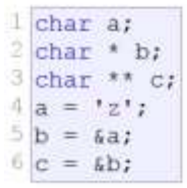
Relational operators (<, >=, etc.) can be used to compare addresses in pointers

Comparing addresses <u>in</u> pointers is not the same as comparing contents <u>pointed at by</u> pointers:

```
if (ptr1 == ptr2)   // compares
                    // addresses

if (*ptr1 == *ptr2) // compares
                    // contents
```

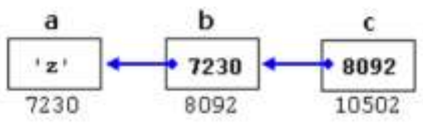
Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). The syntax simply requires an asterisk (*) for each level of indirection in the declaration of the pointer:



```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

This, assuming the randomly chosen memory locations for each variable of 7230, 8092, and 10502, could be represented as:



9.7

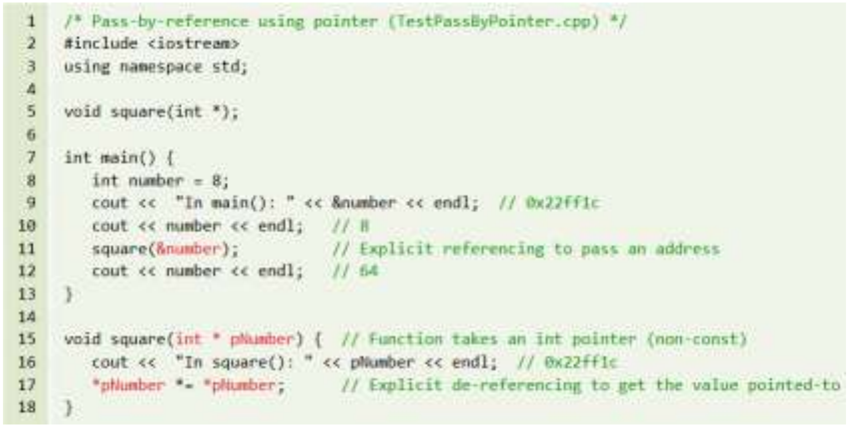
Pointers as Function Parameters

Pass-By-Reference into Functions with Reference Arguments vs. Pointer Arguments

In C/C++, by default, arguments are passed into functions *by value* (except arrays which is treated as pointers). That is, a clone copy of the argument is made and passed into the function. Changes to the clone copy inside the function has no effect to the original argument in the caller. In other words, the called function has no access to the variables in the caller.

Pass-by-Reference with Pointer Arguments

In many situations, we may wish to modify the original copy directly. This can be done by passing a pointer of the object into the function, known as *pass-by-reference*.

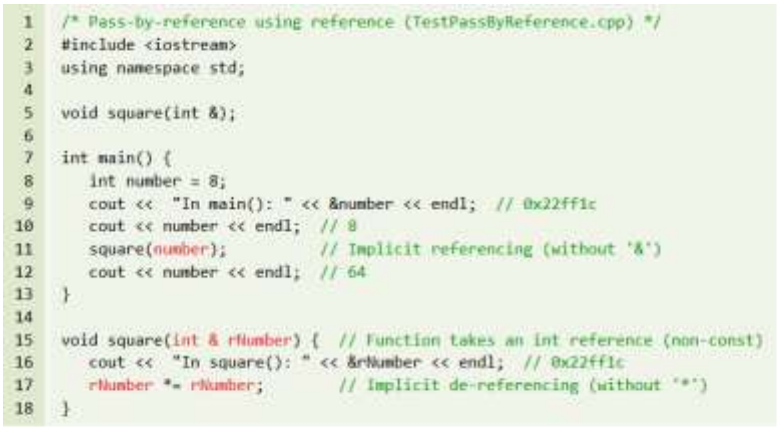


```
/* Pass-by-reference using pointer (TestPassByPointer.cpp) */
#include <iostream>
using namespace std;

void square(int *);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;   // 8
   square(&number);          // Explicit referencing to pass an address
   cout << number << endl;   // 64
}

void square(int * pNumber) {  // Function takes an int pointer (non-const)
   cout <<  "In square(): " << pNumber << endl;  // 0x22ff1c
   *pNumber *= *pNumber;      // Explicit de-referencing to get the value pointed-to
}
```

The called function operates on the same address, and can thus modify the variable in the caller.

Pass-by-Reference with Reference Arguments

Instead of passing pointers into function, you could also pass references into function, to avoid the clumsy syntax of referencing and dereferencing.



```
/* Pass-by-reference using reference (TestPassByReference.cpp) */
#include <iostream>
using namespace std;

void square(int &);

int main() {
   int number = 8;
   cout <<  "In main(): " << &number << endl;  // 0x22ff1c
   cout << number << endl;  // 8
   square(number);          // Implicit referencing (without '&')
   cout << number << endl;  // 64
}

void square(int & rNumber) {  // Function takes an int reference (non-const)
   cout <<  "In square(): " << &rNumber << endl;  // 0x22ff1c
   rNumber *= rNumber;        // Implicit de-referencing (without '*')
}
```

Take note referencing (in the caller) and dereferencing (in the function) are done implicitly. The only coding difference with pass-by-value is in the function's parameter declaration.

Example

```
void swap(int &x, int &y)
{     int temp;
      temp = x;
      x = y;
      y = temp;
}

int num1 = 2, num2 = -3;
swap( num1,  num2);
```

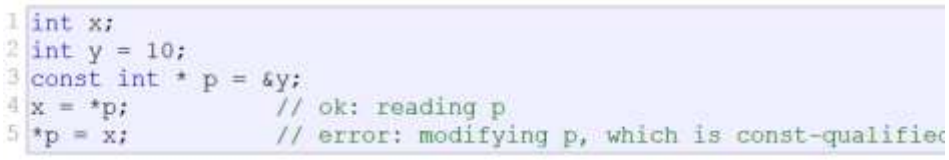
Example

```
void swap(int *x, int *y)
{     int temp;
      temp = *x;
      *x = *y;
      *y = temp;
}

int num1 = 2, num2 = -3;
swap(&num1, &num2);
```

```
#include <iostream>
using namespace std;
void swap(int &x, int &y);
void swap(int *x, int *y);
int main()
{
   int num1 = 2, num2 = -3;
   cout << "num1= " << num1 << " num2= " << num2 <<endl;
   swap(num1, num2);
   cout << "num1= " << num1 << " num2= " << num2 <<endl<<endl;

   cout << "num1= " << num1 << " num2= " << num2 <<endl;
   swap(&num1, &num2);
   cout << "num1= " << num1 << " num2= " << num2 <<endl;

   return 0;
}

void swap(int &x, int &y)          void swap(int *x, int *y)
{           int temp;              {           int temp;
            temp = x;                          temp = *x;
            x = y;                             *x = *y;
            y = temp;                          *y = temp;
}                                  }
```

Pointers and const

Pointers can be used to access a variable by its address, and this access may include modifying the value pointed. But it is also possible to declare pointers that can access the pointed value to read it, but not to modify it. For this, it is enough with qualifying the type pointed to by the pointer as const. For example:



```
int x;
int y = 10;
const int * p = &y;
x = *p;          // ok: reading p
*p = x;          // error: modifying p, which is const-qualifie
```

Here p points to a variable, but points to it in a const-qualified manner, meaning that it can read the value pointed, but it cannot modify it.

One of the use cases of pointers to const elements is as function parameters: a function that takes a pointer to non-const as parameter can modify the value passed as argument, while a function that takes a pointer to const as parameter cannot.

Dynamic Memory

cplusplus.com

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts:

•**The stack:** All variables declared inside the function will take up memory from the stack.

•**The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory previously allocated by new operator.

Getting the Address of a Variable

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

Operators new and new[]

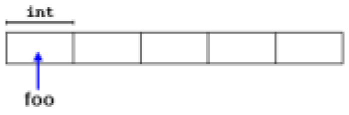
Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets [].

It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

```
pointer = new type
pointer = new type [number_of_elements]
```

```
int * foo;
foo = new int [5];
```

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid block of memory with space for five elements of type int.



Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0] or the expression *foo (both are equivalent).

The second element can be accessed either with foo[1] or *(foo+1), and so on...

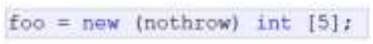
There are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

C++ provides two standard mechanisms to check if the allocation was successful:

This exception method is the method used by default by new, and is the one used in a declaration like:

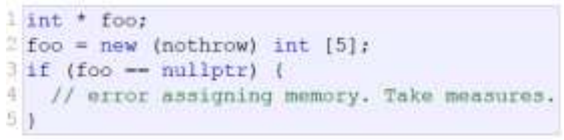
```
foo = new int [5];  // if allocation fails, an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a *null pointer*, and the program continues its execution normally.



```
foo = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:



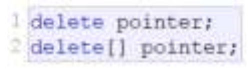
```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // error assigning memory. Take measures.
}
```

What does this code do?

```
int * foo;
foo = new int(5);
```

Operators delete and delete[]

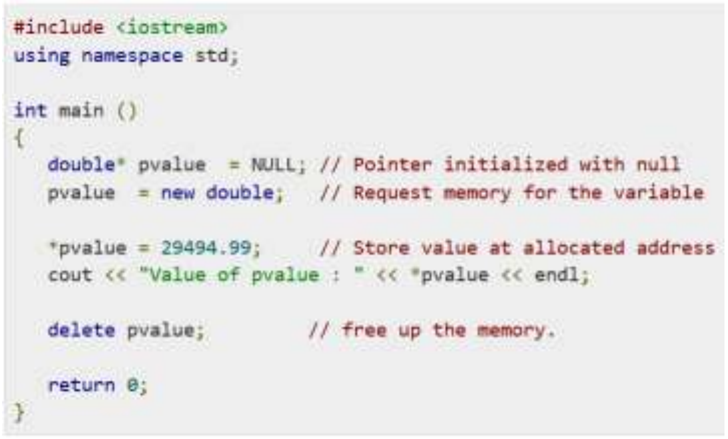
In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:



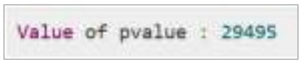
```
delete pointer;
delete[] pointer;
```

The first statement releases the memory of a single element allocated using new,

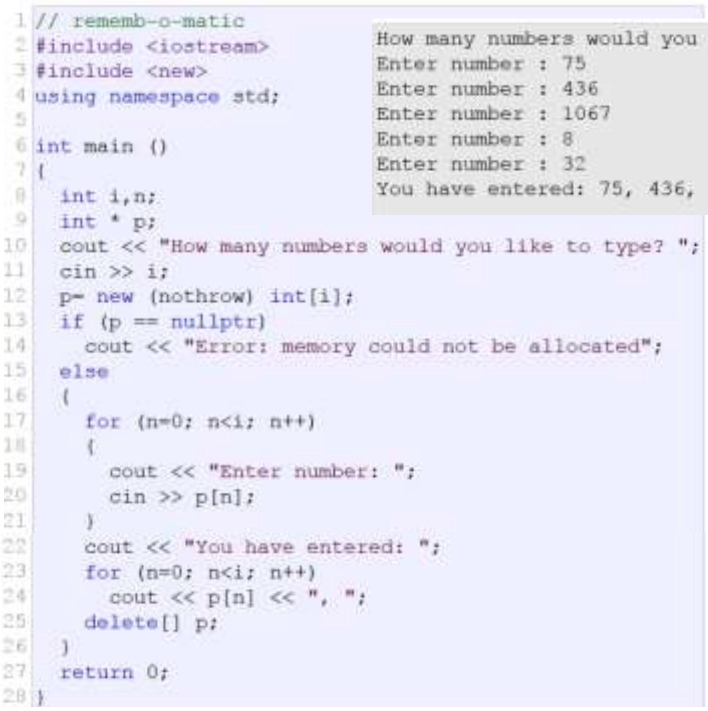
and the second one releases the memory allocated for arrays of elements using new and a size in brackets [].



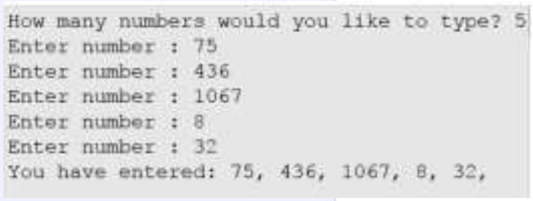
```
#include <iostream>
using namespace std;

int main ()
{
   double* pvalue  = NULL; // Pointer initialized with null
   pvalue  = new double;   // Request memory for the variable

   *pvalue = 29494.99;     // Store value at allocated address
   cout << "Value of pvalue : " << *pvalue << endl;

   delete pvalue;         // free up the memory.

   return 0;
}
```



```
Value of pvalue : 29495
```

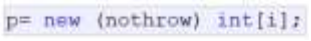


```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
```



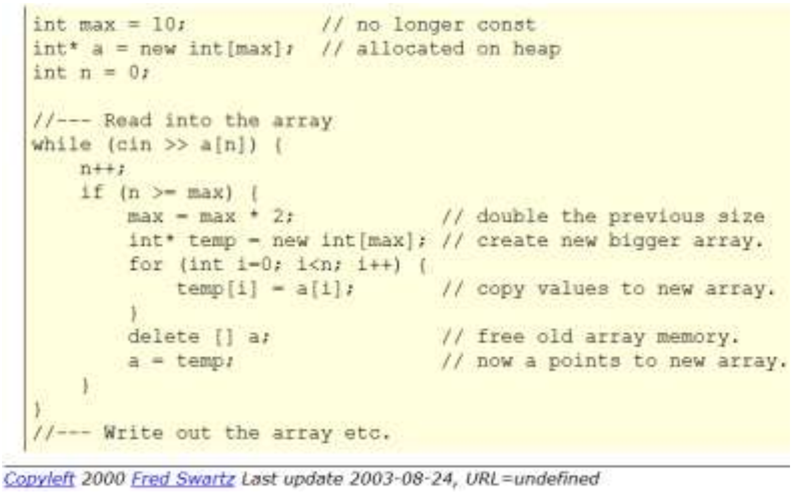
```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:



```
p= new (nothrow) int[i];
```

This program is only limited by the amount of memory. It starts with a small array and expands it only if necessary.



```
int max = 10;           // no longer const
int* a = new int[max];  // allocated on heap
int n = 0;

//--- Read into the array
while (cin >> a[n]) {
    n++;
    if (n >= max) {
        max = max * 2;            // double the previous size
        int* temp = new int[max]; // create new bigger array.
        for (int i=0; i<n; i++) {
            temp[i] = a[i];       // copy values to new array.
        }
        delete [] a;              // free old array memory.
        a = temp;                 // now a points to new array.
    }
}
//--- Write out the array etc.
```

Copyleft 2000 Fred Swartz Last update 2003-08-24, URL=undefined

dynamic 2D array in C++

https://gsamaras.wordpress.com/code/dynamic-2d-array-in-c/

Following is the syntax of new operator for a multi-dimensional array as follows:

```
int ROW = 2;
int COL = 3;
double **pvalue  = new double* [ROW]; // Allocate memory for rows

// Now allocate memory for columns
for(int i = 0; i < COL; i++) {
    pvalue[i] = new double[COL];
}
```

The syntax to release the memory for multi-dimensional will be as follows:

```
for(int i = 0; i < COL; i++) {
    delete[] pvalue[i];
}
delete [] pvalue;
```

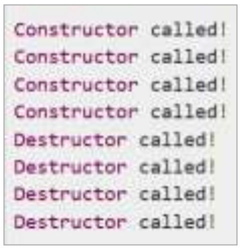
Dynamic Memory Allocation for Objects:

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

```
class Box
{
   public:
      Box() {
         cout << "Constructor called!" <<endl;
      }
      ~Box() {
         cout << "Destructor called!" <<endl;
      }
};

int main( )
{
   Box* myBoxArray = new Box[4];

   delete [] myBoxArray; // Delete array

   return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result:



```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```

Pointers to Objects

Later
next 7 slides hidden

You can also define pointers to class objects. For example, the following statement defines a pointer variable named rectPtr:

```
Rectangle *rectPtr;
```

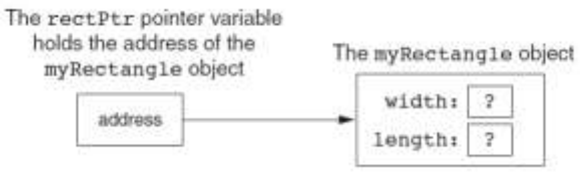
The rectPtr variable is not an object, but it can hold the address of a Rectangle object.

The following code shows an example.

```
Rectangle myRectangle;        // A Rectangle object
Rectangle *rectPtr;           // A Rectangle pointer
rectPtr = &myRectangle;       // rectPtr now points to myRectangle
```

The first statement creates a Rectangle object named myRectangle. The second statement creates a Rectangle pointer named rectPtr. The third statement stores the address of the myRectangle object in the rectPtr pointer.

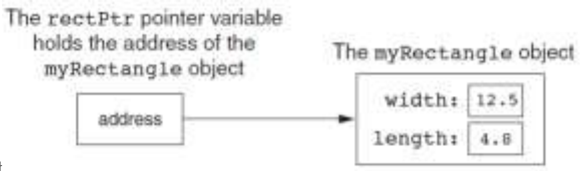
```
Rectangle myRectangle;        // A Rectangle object
Rectangle *rectPtr;           // A Rectangle pointer
rectPtr = &myRectangle;       // rectPtr now points to myRectangle
```


The rectPtr pointer variable holds the address of the myRectangle object
The myRectangle object
address
width: ?
length: ?

The rectPtr pointer can then be used to call member functions by using the

-> operator.

```
rectPtr->setWidth(12.5);
rectPtr->setLength(4.8);
```


The rectPtr pointer variable holds the address of the myRectangle object
The myRectangle object
address
width: 12.5
length: 4.8

Class object pointers can be used to dynamically allocate objects.

```
 1  // Define a Rectangle pointer.
 2  Rectangle *rectPtr;
 3
 4  // Dynamically allocate a Rectangle object.
 5  rectPtr = new Rectangle;
 6
 7  // Store values in the object's width and length.
 8  rectPtr->setWidth(10.0);
 9  rectPtr->setLength(15.0);
10
11  // Delete the object from memory.
12  delete rectPtr;
13  rectPtr = 0;
```

Line 2 defines rectPtr as a Rectangle pointer.

Line 5 uses the **new** operator to dynamically allocate a Rectangle object and assign its address to rectPtr.

Lines 8 and 9 store values in the dynamically allocated object's width and length variables.

Line 12 deletes the object from memory and line 13 stores the address 0 in rectPtr.

```
//*******************************************************
// Function main                                        *
//*******************************************************

int main()
{
   double number;        // To hold a number
   double totalArea;     // The total area
   Rectangle *kitchen;   // To point to kitchen dimensions
   Rectangle *bedroom;   // To point to bedroom dimensions
   Rectangle *den;       // To point to den dimensions

   // Dynamically allocate the objects.
   kitchen = new Rectangle;
   bedroom = new Rectangle;
   den = new Rectangle;

   // Get the kitchen dimensions.
   cout << "What is the kitchen's length? ";
   cin >> number;                            // Get the length
   kitchen->setLength(number);               // Store in kitchen object
   cout << "What is the kitchen's width? ";
   cin >> number;                            // Get the width
   kitchen->setWidth(number);                // Store in kitchen object
```

```

    // Get the bedroom dimensions.
    cout << "What is the bedroom's length? ";
    cin >> number;                           // Get the length
    bedroom->setLength(number);              // Store in bedroom object
    cout << "What is the bedroom's width? ";
    cin >> number;                           // Get the width
    bedroom->setWidth(number);               // Store in bedroom object

    // Get the den dimensions.
    cout << "What is the den's length? ";
    cin >> number;                           // Get the length
    den->setLength(number);                  // Store in den object
    cout << "What is the den's width? ";
    cin >> number;                           // Get the width
    den->setWidth(number);                   // Store in den object

    // Calculate the total area of the three rooms.
    totalArea = kitchen->getArea() + bedroom->getArea()
              + den->getArea();

```

```
    // Display the total area of the three rooms.
    cout << "The total area of the three rooms is "
         << totalArea << endl;

    // Delete the objects from memory.
    delete kitchen;
    delete bedroom;
    delete den;
    kitchen = 0;    // Make kitchen point to null.
    bedroom = 0;    // Make bedroom point to null.
    den = 0;        // Make den point to null.

    return 0;
}
```

```
What is the kitchen's length? 10 [Enter]
What is the kitchen's width? 14 [Enter]
What is the bedroom's length? 15 [Enter]
What is the bedroom's width? 12 [Enter]
What is the den's length? 20 [Enter]
What is the den's width? 30 [Enter]
The total area of the three rooms is 920
```

Array of pointers

There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX];
```

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

tutorialspoint.com/cplusplus

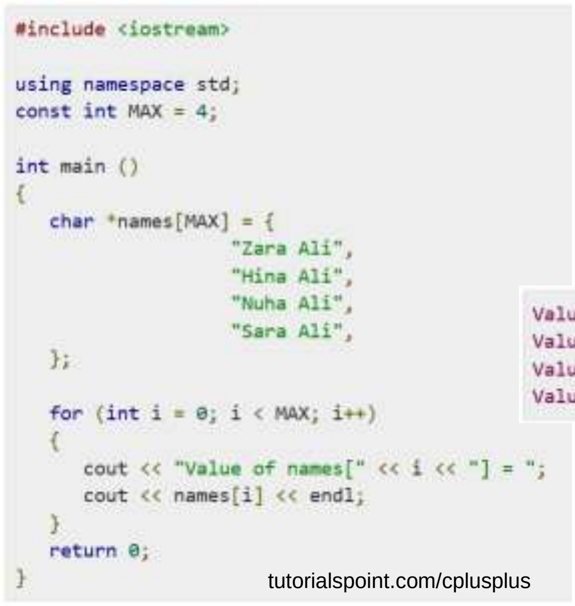
Following example makes use of three integers which will be stored in an array of pointers as follows:

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};
   int *ptr[MAX];

   for (int i = 0; i < MAX; i++)
   {
      ptr[i] = &var[i]; // assign the address of integer.
   }
   for (int i = 0; i < MAX; i++)
   {
      cout << "Value of var[" << i << "] = ";
      cout << *ptr[i] << endl;
   }
   return 0;
}
```

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

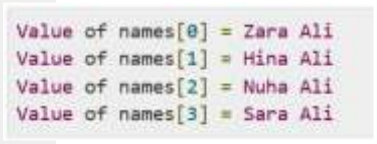
tutorialspoint.com/cplusplus

You can also use an array of pointers to character to store a list of strings as follows:



```
#include <iostream>

using namespace std;
const int MAX = 4;

int main ()
{
   char *names[MAX] = {
                   "Zara Ali",
                   "Hina Ali",
                   "Nuha Ali",
                   "Sara Ali",
   };

   for (int i = 0; i < MAX; i++)
   {
      cout << "Value of names[" << i << "] = ";
      cout << names[i] << endl;
   }
   return 0;
}
```

tutorialspoint.com/cplusplus



```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

Dynamically allocated array of objects

```
Polygon * polyArrPtr;              // a pointer to an object
polyArrPtr = new Polygon[5];       // allocate memory
polyArrPtr[0]= &poly1;
polyArrPtr[1]= &poly2;

polyArrPtr[0]->displayData();
polyArrPtr[0]->drawPolygon();
polyArrPtr[1]->displayData();
polyArrPtr[1]->drawPolygon();

delete [] polyArrPtr;
```

- Use member-selection (->) operator to refer to its class members, e.g. obj->memberName

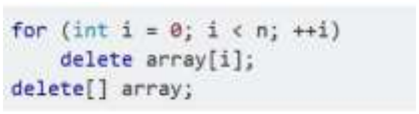
Dynamically allocated array of pointers

```
Polygon ** polyArrPtr;                // a pointer to an array of pointers
polyArrPtr = new Polygon * [5];   // allocate memory
polyArrPtr[0]= &poly1;
polyArrPtr[1]= &poly2;

polyArrPtr[0]->displayData();
polyArrPtr[0]->drawPolygon();
polyArrPtr[1]->displayData();
polyArrPtr[1]->drawPolygon();

for (int i=0; i<5; i++)
  delete polyPtr[i]; //delete the objects
delete [] polyArrPtr; // delete the array of pointers
```

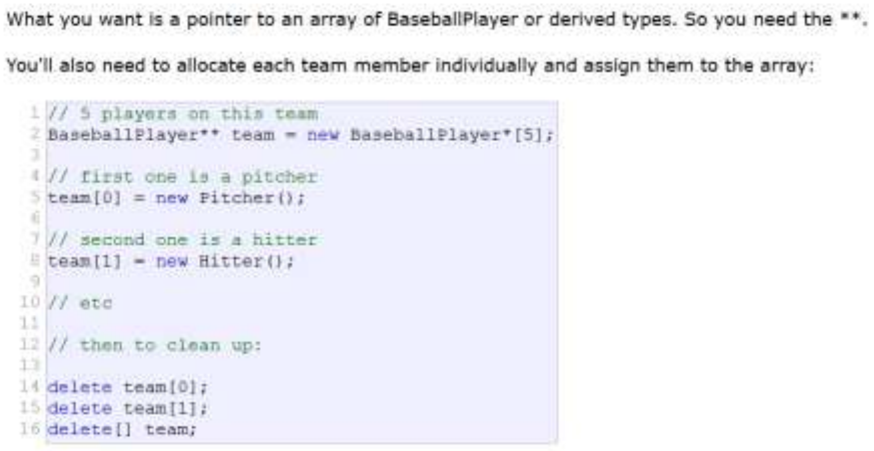
- Use member-selection (->) operator to refer to its class members, e.g. obj->memberName

if I have an array that contains pointers to other objects, how do I deallocate the memory referenced by each pointer in the array AND the array itself?

Loop through the array to delete every object and then delete the array:



```
for (int i = 0; i < n; ++i)
    delete array[i];
delete[] array;
```



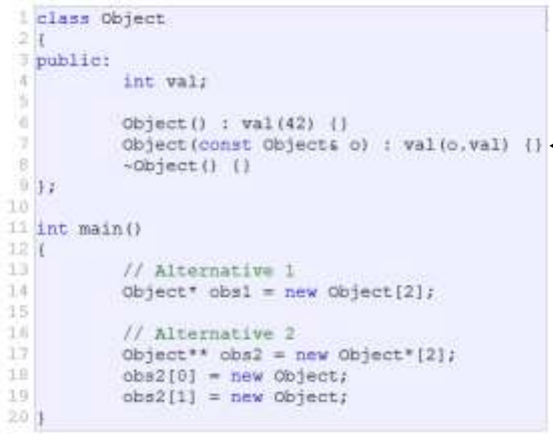
What you want is a pointer to an array of BaseballPlayer or derived types. So you need the **.

You'll also need to allocate each team member individually and assign them to the array:

```
// 5 players on this team
BaseballPlayer** team = new BaseballPlayer*[5];

// first one is a pitcher
team[0] = new Pitcher();

// second one is a hitter
team[1] = new Hitter();

// etc

// then to clean up:

delete team[0];
delete team[1];
delete[] team;
```

Array of objects vs. array of pointers

Consider the following code:



```
class Object
{
public:
        int val;

        Object() : val(42) {}
        Object(const Object& o) : val(o.val) {}
        ~Object() {}
};

int main()
{
        // Alternative 1
        Object* obs1 = new Object[2];

        // Alternative 2
        Object** obs2 = new Object*[2];
        obs2[0] = new Object;
        obs2[1] = new Object;
}
```

← Copy constructor

Do these two do the same thing?

The first allocates 2 objects on the heap,

The second allocates 2 pointers and 2 objects on the heap.

Destructors and Dynamically Allocated Class Objects

If a class object has been dynamically allocated by the new operator, its memory should be released when the object is no longer needed. For example, in the following code objectPtr is a pointer to a dynamically allocated InventoryItem class object.

```
// Define an InventoryItem pointer.
InventoryItem *objectPtr;
// Dynamically create an InventoryItem object.
objectPtr = new InventoryItem("Wrench", 8.75, 20);
```

Destructors and Dynamically Allocated Class Objects

The following statement shows the delete operator being used to destroy the dynamically created object.

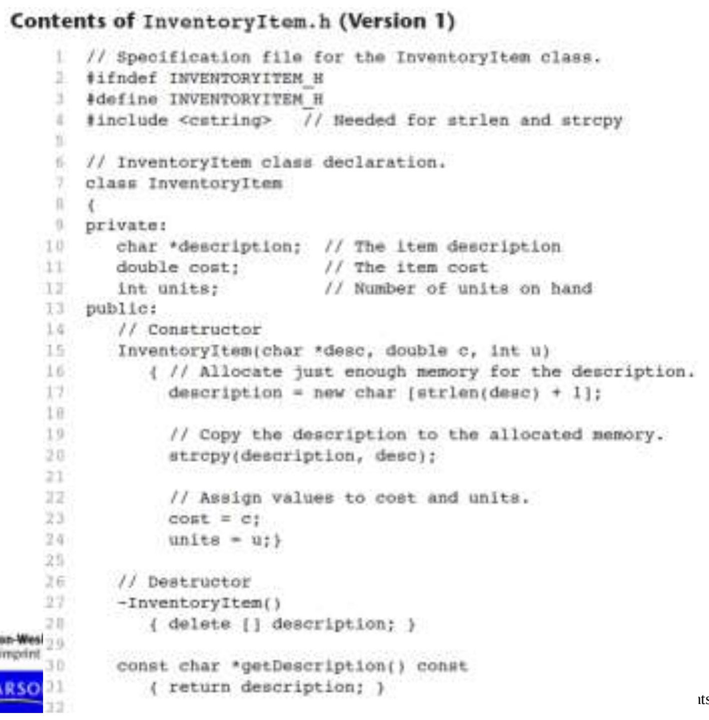
```
delete objectPtr;
```

When the object pointed to by objectPtr is destroyed, its destructor is automatically called.

The following code shows a more practical example of a class with a destructor. The InventoryItem class holds the following data about an item that is stored in inventory:

- The item's description
- The item's cost
- The number of units in inventory

The constructor accepts arguments for all three items. The description is passed as a pointer to a C-string. Rather than storing the description in a char array with a fixed size, the constructor gets the length of the C-string and dynamically allocates just enough memory to hold it. The destructor frees the allocated memory when the object is destroyed.



Contents of InventoryItem.h (Version 1)

```
// Specification file for the InventoryItem class.
#ifndef INVENTORYITEM_H
#define INVENTORYITEM_H
#include <cstring>   // Needed for strlen and strcpy

// InventoryItem class declaration.
class InventoryItem
{
private:
   char *description;  // The item description
   double cost;        // The item cost
   int units;          // Number of units on hand
public:
   // Constructor
   InventoryItem(char *desc, double c, int u)
      { // Allocate just enough memory for the description.
        description = new char [strlen(desc) + 1];

        // Copy the description to the allocated memory.
        strcpy(description, desc);

        // Assign values to cost and units.
        cost = c;
        units = u;}

   // Destructor
   ~InventoryItem()
      { delete [] description; }

   const char *getDescription() const
      { return description; }

   double getCost() const
      { return cost; }

   int getUnits() const
      { return units; }
};
#endif
```

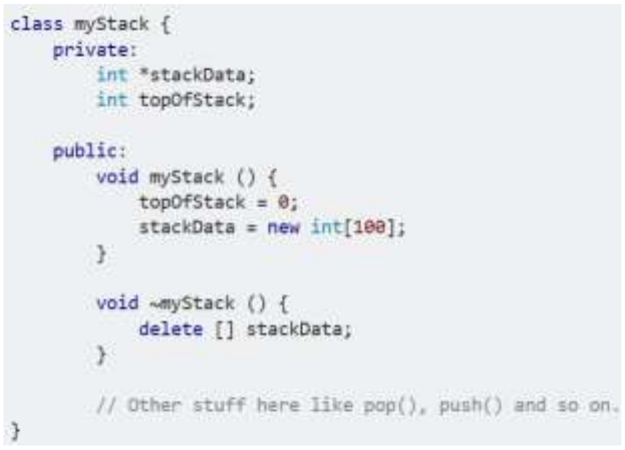
Program 13-11

```
// This program demonstrates a class with a destructor.
#include <iostream>
#include <iomanip>
#include "InventoryItem.h"
using namespace std;

int main()
{
   // Define an InventoryItem object with the following data:
   // Description: Wrench  Cost: 8.75   Units on hand: 20
   InventoryItem stock("Wrench", 8.75, 20);

   // Set numeric output formatting.
   cout << setprecision(2) << fixed << showpoint;

   // Display the object's data.
   cout << "Item Description: " << stock.getDescription() << endl;
   cout << "Cost: $" << stock.getCost() << endl;
   cout << "Units on hand: " << stock.getUnits() << endl;
   return 0;
}
```

Program Output

```
Item Description: Wrench
Cost: $8.75
Units on hand: 20
```

delete p does two things: it calls the destructor and it deallocates the memory.
If delete deallocates the memory, then what's the need of the destructor?



```
class myStack {
    private:
        int *stackData;
        int topOfStack;

    public:
        void myStack () {
            topOfStack = 0;
            stackData = new int[100];
        }

        void ~myStack () {
            delete [] stackData;
        }

        // Other stuff here like pop(), push() and so on.
}
```

In the above example, think of what would happen if the destructor was not called every time one of your stacks got deleted. There is no automatic garbage collection in C++ in this case so the stackData memory would leak and you'd eventually run out.

Destructors and how to use them?

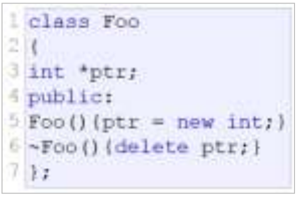
So I've made some pointers using the *new* keyword and I know that every time I use the new keyword I must use the *delete* keyword to deallocate that memory. My question is, if I've created a pointer like so:

```
SomeType *  myFunction() {
            SomeType *newPtr = new SomeType;
            return newPtr;
}
```

Can I deallocate this memory in one step in the destructor?

Just call: `delete newPtr;` When you want to delete it.

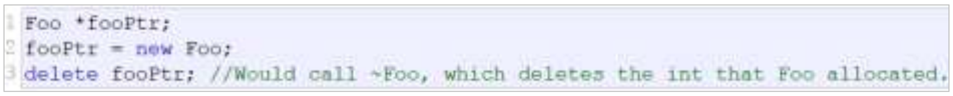
If newPtr is a member of a class then yeah, stick the delete inside the



```
class Foo
{
int *ptr;
public:
Foo(){ptr = new int;}
~Foo(){delete ptr;}
};
```

http://www.cplusplus.com/forum

Remember that if you have a pointer to a class, calling delete on that pointer will call the destructor of the class it is pointing to, which gives you a chance to delete any dynamic memory that class itself may have allocated. So this:



```
Foo *fooPtr;
fooPtr = new Foo;
delete fooPtr; //Would call ~Foo, which deletes the int that Foo allocated.
```

http://www.cplusplus.com/forum

9.9

Returning Pointers from Functions

Returning Pointers from Functions

Pointer can be the return type of a function:

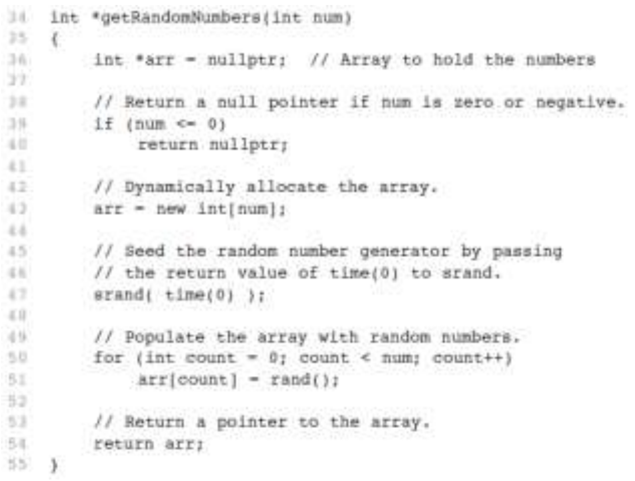
```
int* newNum();
```

The function must not return a pointer to a local variable in the function.

A function should only return a pointer:

    to data that was passed to the function as an argument, or
    to dynamically allocated memory

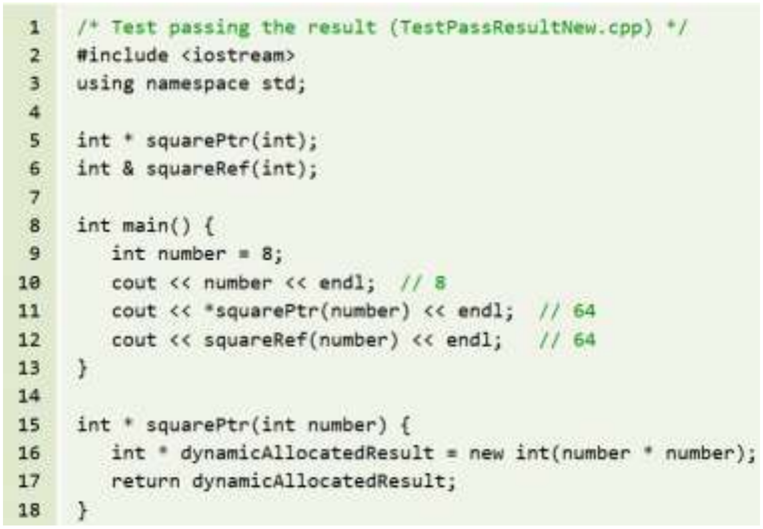
From Program 9-15



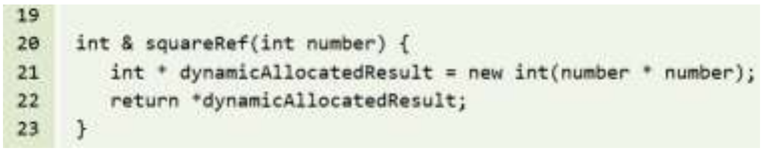
```
int *getRandomNumbers(int num)
{
    int *arr = nullptr;  // Array to hold the numbers

    // Return a null pointer if num is zero or negative.
    if (num <= 0)
        return nullptr;

    // Dynamically allocate the array.
    arr = new int[num];

    // Seed the random number generator by passing
    // the return value of time(0) to srand.
    srand( time(0) );

    // Populate the array with random numbers.
    for (int count = 0; count < num; count++)
        arr[count] = rand();

    // Return a pointer to the array.
    return arr;
}
```

Passing Dynamically Allocated Memory as Return Value by Reference

Instead, you need to dynamically allocate a variable for the return value, and return it by reference.



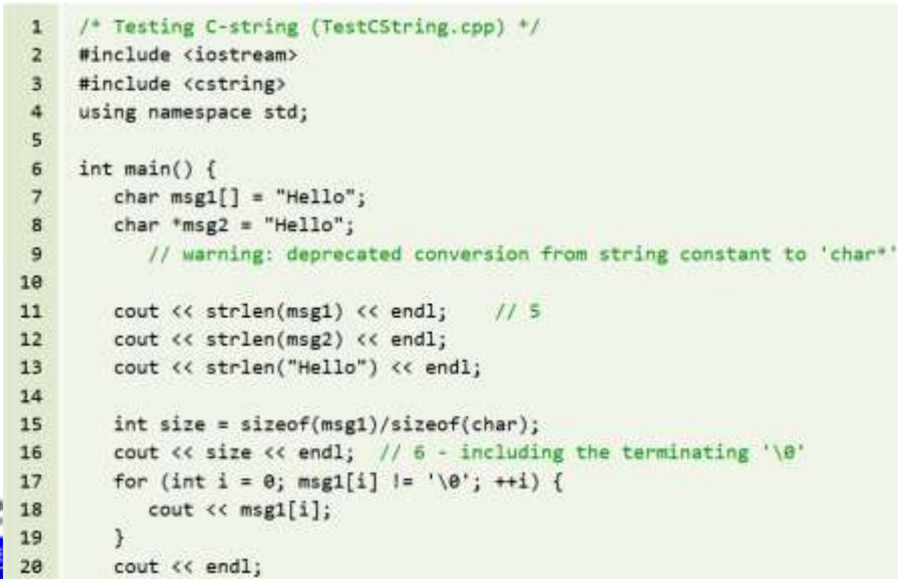
```cpp
/* Test passing the result (TestPassResultNew.cpp) */
#include <iostream>
using namespace std;

int * squarePtr(int);
int & squareRef(int);

int main() {
   int number = 8;
   cout << number << endl;  // 8
   cout << *squarePtr(number) << endl;  // 64
   cout << squareRef(number) << endl;   // 64
}

int * squarePtr(int number) {
   int * dynamicAllocatedResult = new int(number * number);
   return dynamicAllocatedResult;
}
```



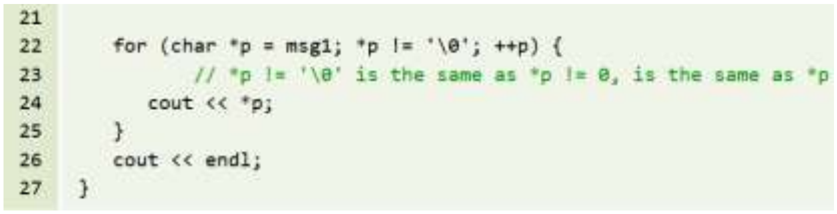
```cpp

int & squareRef(int number) {
   int * dynamicAllocatedResult = new int(number * number);
   return *dynamicAllocatedResult;
}
```

C-String and Pointer

C-string (of the C language) is a character array, terminated with a null character '\0'. For example,

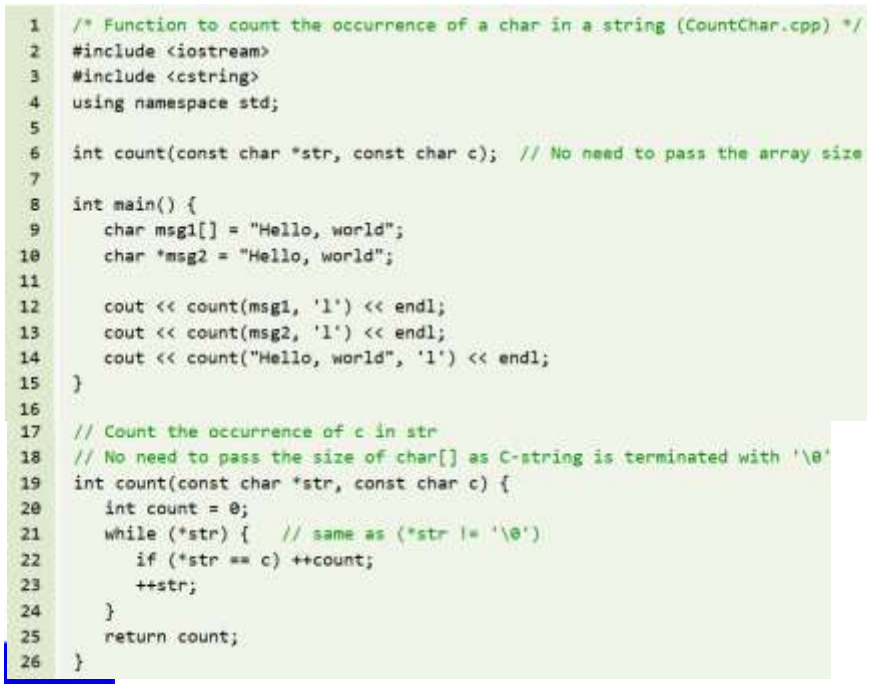


```
/* Testing C-string (TestCString.cpp) */
#include <iostream>
#include <cstring>
using namespace std;

int main() {
   char msg1[] = "Hello";
   char *msg2 = "Hello";
      // warning: deprecated conversion from string constant to 'char*'

   cout << strlen(msg1) << endl;    // 5
   cout << strlen(msg2) << endl;
   cout << strlen("Hello") << endl;

   int size = sizeof(msg1)/sizeof(char);
   cout << size << endl;  // 6 - including the terminating '\0'
   for (int i = 0; msg1[i] != '\0'; ++i) {
      cout << msg1[i];
   }
   cout << endl;
```



```

   for (char *p = msg1; *p != '\0'; ++p) {
         // *p != '\0' is the same as *p != 0, is the same as *p
      cout << *p;
   }
   cout << endl;
}
```

Take note that for C-String function such as strlen() (in header cstring, ported over from C's string.h), there is no need to pass the array length into the function. This is because C-Strings are terminated by '\0'. The function can iterate thru the characters in the array until '\0'.



```
/* Function to count the occurrence of a char in a string (CountChar.cpp) */
#include <iostream>
#include <cstring>
using namespace std;

int count(const char *str, const char c);  // No need to pass the array size

int main() {
   char msg1[] = "Hello, world";
   char *msg2 = "Hello, world";

   cout << count(msg1, 'l') << endl;
   cout << count(msg2, 'l') << endl;
   cout << count("Hello, world", 'l') << endl;
}

// Count the occurrence of c in str
// No need to pass the size of char[] as C-string is terminated with '\0'
int count(const char *str, const char c) {
   int count = 0;
   while (*str) {   // same as (*str != '\0')
      if (*str == c) ++count;
      ++str;
   }
   return count;
}
```

Checkpoint

9.9 Assuming `arr` is an array of `int`s, will each of the following program segments display "True" or "False"?

A)
```
if (arr < &arr[1])
   cout << "True";
else
   cout << "False";
```

B)
```
if (&arr[4] < &arr[1])
   cout << "True";
else
   cout << "False";
```

C)
```
if (arr != &arr[2])
   cout << "True";
else
   cout << "False";
```

D)
```
if (arr != &arr[0])
   cout << "True";
else
   cout << "False";
```

9.11 Complete the following program skeleton. When finished, the program will ask the user for a length (in inches), convert that value to centimeters, and display the result. You are to write the function `convert`. (*Note:* 1 inch = 2.54 cm. Do not modify function main.)

```
#include <iostream>
#include <iomanip>
using namespace std;

// Write your function prototype here.

int main()
{
   double measurement;

   cout << "Enter a length in inches, and I will convert\n";
   cout << "it to centimeters: ";
   cin >> measurement;
   convert(&measurement);
   cout << fixed << setprecision(4);
   cout << "Value in centimeters: " << measurement << endl;
   return 0;
}
//
// Write the function convert here.
//
```

9.12 Look at the following array definition:

```
const int numbers[SIZE] = { 18, 17, 12, 14 };
```

Suppose we want to pass the array to the function `processArray` in the following manner:

```
processArray(numbers, SIZE);
```

Which of the following function headers is the correct one for the `processArray` function?

A) `void processArray(const int *arr, int size)`

B) `void processArray(int * const arr, int size)`

9.13 Assume `ip` is a pointer to an `int`. Write a statement that will dynamically allocate an integer variable and store its address in `ip`. Write a statement that will free the memory allocated in the statement you wrote above.

9.14 Assume `ip` is a pointer to an `int`. Then, write a statement that will dynamically allocate an array of 500 integers and store its address in `ip`. Write a statement that will free the memory allocated in the statement you just wrote.

Using Object Pointer

- Declare an object pointer:

```
T * pObj;
```

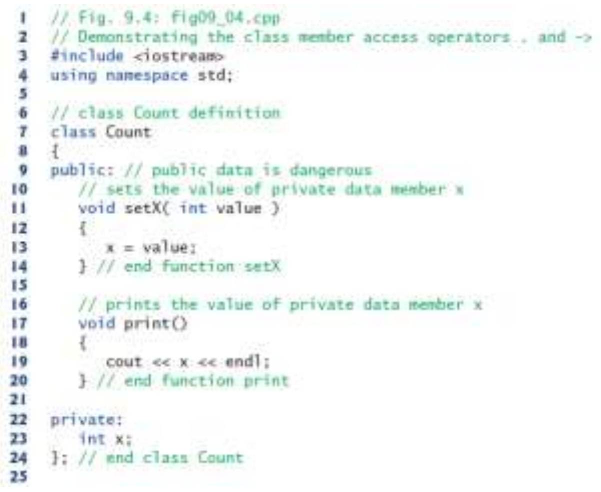
- Either initialize the pointer to an existing object, or dynamically allocate an object.

```
pObj = &obj;
// OR
pObj = new T(...);
```

- Use member-selection (->) operator to refer to its class members, e.g. obj->memberName.

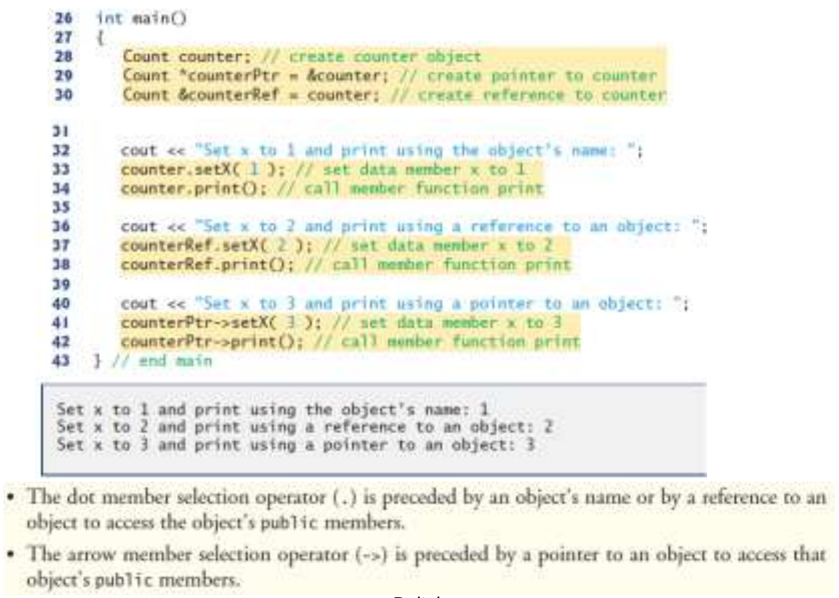
- Use dereferencing (*) operator to get its content, e.g., *obj.

ntu.edu.sg



```
// Fig. 9.4: fig09_04.cpp
// Demonstrating the class member access operators . and ->
#include <iostream>
using namespace std;

// class Count definition
class Count
{
public: // public data is dangerous
   // sets the value of private data member x
   void setX( int value )
   {
      x = value;
   } // end function setX

   // prints the value of private data member x
   void print()
   {
      cout << x << endl;
   } // end function print

private:
   int x;
}; // end class Count

```

Deitel



```
int main()
{
   Count counter; // create counter object
   Count *counterPtr = &counter; // create pointer to counter
   Count &counterRef = counter; // create reference to counter

   cout << "Set x to 1 and print using the object's name: ";
   counter.setX( 1 ); // set data member x to 1
   counter.print(); // call member function print

   cout << "Set x to 2 and print using a reference to an object: ";
   counterRef.setX( 2 ); // set data member x to 2
   counterRef.print(); // call member function print

   cout << "Set x to 3 and print using a pointer to an object: ";
   counterPtr->setX( 3 ); // set data member x to 3
   counterPtr->print(); // call member function print
} // end main
```

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

- The dot member selection operator (.) is preceded by an object's name or by a reference to an object to access the object's public members.
- The arrow member selection operator (->) is preceded by a pointer to an object to access that object's public members.

Deitel

Keyword "this"

You can use keyword "this" to refer to *this* instance inside a class definition.

One of the main usage of keyword this is to resolve ambiguity between the names of data member and function parameter. For example,

```
class Circle {
private:
   double radius;                 // Member variable called "radius"
   ......
public:
   void setRadius(double radius) { // Function's argument also called "radi
      this->radius = radius;
         // "this.radius" refers to this instance's member variable
         // "radius" resolved to the function's argument.
   }
   ......
}
```

ntu.edu.sg

Alternatively, you could use a prefix (such as m_) or suffix (such as _) to name the data members to avoid name clashes. For example,

```
class Circle {
private:
   double m_radius;  // or radius_
   ......
public:
   void setRadius(double radius) {
      m_radius = radius;  // or radius_ = radius
   }
   ......
}
```

ntu.edu.sg

9.10

Using Smart Pointers to Avoid Memory Leaks

Using Smart Pointers to Avoid Memory Leaks

In C++ 11, you can use *smart pointers* to dynamically allocate memory and not worry about deleting the memory when you are finished using it.

Three types of smart pointer:

```
unique_ptr
shared_ptr
weak_ptr
```

Must `#include` the memory header file:

```
#include <memory>
```

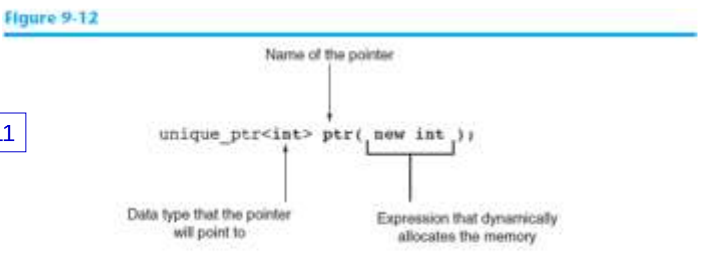
In this book, we introduce `unique_ptr`:

```
unique_ptr<int> ptr( new int );
```

Using Smart Pointers to Avoid Memory Leaks

C++11

Figure 9-12


Name of the pointer
unique_ptr<int> ptr(new int);
Data type that the pointer will point to
Expression that dynamically allocates the memory

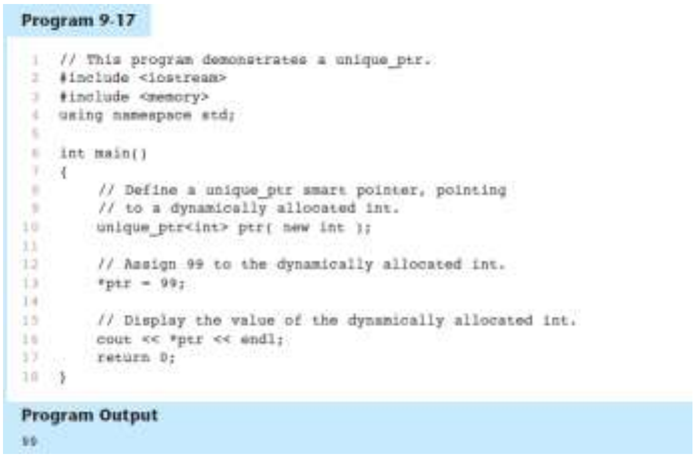
The notation `<int>` indicates that the pointer can point to an `int`.

The name of the pointer is `ptr`.

The expression `new int` allocates a chunk of memory to hold an `int`.

The address of the chunk of memory will be assigned to `ptr`.

Using Smart Pointers in Program 9-17



Program 9-17

```
// This program demonstrates a unique_ptr.
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    // Define a unique_ptr smart pointer, pointing
    // to a dynamically allocated int.
    unique_ptr<int> ptr( new int );

    // Assign 99 to the dynamically allocated int.
    *ptr = 99;

    // Display the value of the dynamically allocated int.
    cout << *ptr << endl;
    return 0;
}
```

Program Output

```
99
```