C++ Basics

Rosemary Monahan
CS613 OO&C++
September 2002

Introduction to Basic Procedural Programming (with C++)

Try to simulate **compilation** and **execution** in your head

Try different variations on a theme - simply edit the examples

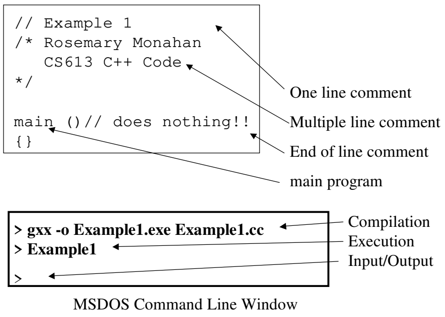
Easiest way to learn programming is to experiment

The computer is a laboratory

If you don't know something then write a program to give you the answer … if you can

Otherwise, try looking in a book (or on the web)

You can always ask someone (doesn't have to be me)



```
// Example 1
/* Rosemary Monahan
   CS613 C++ Code
*/

main ()// does nothing!!
{}
```

One line comment

Multiple line comment

End of line comment

main program

```
> gxx -o Example1.exe Example1.cc
> Example1
>
```

Compilation

Execution

Input/Output

MSDOS Command Line Window

Example1 Variations - what will happen in each case?

```
/* Example 1A
   CS613 C++ Code
*/

int main ()// does nothing!!
{return 0;}
```

```
/* Example 1B
   CS613 C++ Code
*/

int main ()
{}
```

```
/* Example 1C
   CS613 C++ Code
*/

main ()// does nothing!!
{return 0;}
```

```
// Example 1D
int main () {return 7}
```

```
// Example 1E
int main (){return 0.7;;}
```

```
// Example 1F
float main (){return 0.7;}
```

Example Variations - some notes

Try to learn something new from each variation - see 1B

Try not to change more than one thing at once - see 1C

Try to introduce different compiler errors - syntax and semantics - see 1D

Try to distinguish errors from warnings - see 1E

Try to remember that the compiler can be sneaky - see 1F

Example 2: output to the screen

```
// Example 2

#include <iostream.h>
int main (){cout<<"Hello World!";}
```

Writes (outputs) -
Hello World!
To the screen

Question - what sort of variations might be useful ?

- •remove the `include`
- •change the output `Hello World!`
- •add in a `return`

Variation 2A: remove the include

```
// Example 2

//#include <iostream.h>
int main (){cout<<"Hello World!";}
```

Writes (outputs) -

Example2A.cc: In function `int main()':
Example2A.cc:4: `cout' undeclared (first use this function)
Example2A.cc:4: (Each undeclared identifier is reported only once
Example2A.cc:4: for each function it appears in.)

To the screen

Note: this is a semantic error reported by the compiler

Variation 2B: change the output

```
// Example 2B

#include <iostream.h>
int main (){
cout<<"Hello Universe, ";
cout <<"how are"<<" you"<<"?";}
```

Program contains 2 instructions

You can **stream strings** together

Writes (outputs) -
Hello Universe, how are you?

Note: this is not an error it is the result of the code being executed

Variation 2C: change the output

```
// Example 2C

#include <iostream.h>
int main (){
cout<<"Hello Universe, \n";
cout <<"how are"<<endl<<" you"<<'?';}
```

String, in double quotes, contains special character `\n`

A `?` character between single quotes

Note the use of `endl` - a stream manipulator object!

Writes (outputs) -

Hello Universe,
how are
you?

Note: the format - spaces and newlines

Variation 2D: change the output

```
// Example 2D

#include <iostream.h>
int main (){
cout<<"7 + 8 =  \n";
cout <<7+8;}
```

String, in double quotes, contains special character `\n`

An **integer expression** which needs to be evaluated

Writes (outputs) -

7 + 8 =
15

Note: the addition is correct (in base 10)

Variation 2E: add in a return

```
// Example 2E

#include <iostream.h>
int main (){return 0;
cout<<"7 + 8 =  \n";
cout <<7+8;}
```

Return at the beginning of the main body

Writes (outputs) absolutely nothing to the screen

Note: if we put the return in as the last line then we get the same behaviour as before

Example 3 - introducing variables

```
// Example 3

#include <iostream.h>
int main (){
int x = 7+8; cout <<x;
}
```

The variable `x` is identified by the sequence of characters “x”.
It has type **`int`** (an integer)

`x` is **declared** to be an `int` and **initialised** to value 15 in the same statement.

This outputs -

15

QUESTION: what variations to try?

Variation 3A: What if the variable is not initialised?

```
// Example 3A

#include <iostream.h>
int main (){
int x; cout <<x;
}
```

Output:
686796

Variation 3A: What if the variable is changed?

```
// Example 3B

#include <iostream.h>
int main (){
int x; cout <<x; x = 0;
cout <<x;
}
```

Output:
6867960

Variation 3C: What if the variable is initialised twice?

```
// Example 3C

#include <iostream.h>
int main (){
int x; int x; cout <<x;
}
```

Output:

```
Example3C.cc: In function `int main()':
Example3C.cc:5: redeclaration of `int x'
Example3C.cc:5: `int x' previously declared here
```

Variation 3D: What if the variable is changed?

```
// Example 3D

#include <iostream.h>
int main (){
int x = 1;
x = x+x;cout<<x*x;
}
```

We can change the value of `x` using `x`

This does not change the value of `x`

Output:
4

Already some simple rules to learn -

•variables

•identifiers

•types

•values

•constants

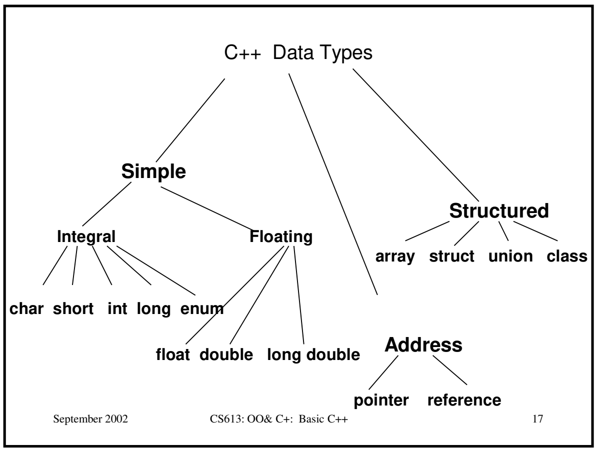
•keywords

Identifiers

- An identifier must start with a letter or underscore, and be followed by zero or more letters
- **C++ is case sensitive**
- VALID

age_of_dog	**TaxRate98**
PrintHeading	**AgeOfHorse**

- INVALID examples - try to find them yourselves using the compiler



Typical size of data types

Type	Number of Bytes
`char`	1
`short (short int)`	2
`int`	4
`unsigned int`	4
`enum`	2
`long (long int)`	4
`float`	4
`double`	8
`long double`	12

Finding size of data type - system dependent results

- Use **`sizeof`** to find the size of a type

  e.g.

  ```
  cout << sizeof(int)
  ```

Finds size of int types
on our system.

Enumerated types: *enum*

Used to define constant values

```
enum days
{ Sunday, Monday, Tuesday,
   Wednesday, Thursday, Friday, Saturday
} yesterday, today;
 days tomorrow;
```

QUESTION: what are the values of `Tuesday, today, tomorrow, yesterday`?

Default values may be overridden in the enum list.

Enumerated types

```
// Example 4

#include <iostream.h>

int main (){
enum days {Sunday, Monday, Tuesday,Wednesday,
           Thursday, Friday, Saturday} yesterday, today;
days tomorrow;
cout << Tuesday<<endl;
cout<< yesterday<<endl<<today<<endl<<tomorrow;
}
```

Outputs:

```
2
686796
143896
117208
```

Boolean type

C++ has a built-in logical or Boolean type

```
// Example 5

#include <iostream.h>

int main (){
bool flag = true;
cout << flag<< endl<<!flag}
```

This outputs:

```
1

0
```

Variation 5a - booleans are really integers?

```
// Example 5a

#include <iostream.h>

int main (){
bool flag1 = 100, flag2 = false, flag3;
cout << flag1<< endl<<flag2<<endl<<flag3;}
```

`true` is any non-zero `int`

`false` is zero

This outputs:

```
1

0

122
```

`true` is output as 1
`false` is output as 0

This was not initialised

Giving a value to a variable

In your program you can assign (give) a value to the variable by using the assignment operator =

```
Age_Of_Dog = 12;
```

or by another standard method, such as

```
cout << “How old is your dog?”;
cin  >> Age_Of_Dog;
```

What is a Named Constant?

- A **named constant** is a location in memory which we can refer to by an identifier, and in which a **data value that cannot be changed** is stored.

VALID CONSTANT DECLARATIONS

- `const   char   STAR  =  '*' ;`
- `const   float   PI  =  3.14159 ;`
- `const   int      MAX_SIZE  =  50 ;`

Note: all caps

keywords: words reserved to C++

- `bool, true, false,`
- `char, float, int, unsigned, double, long`
- `if, else, for, while, switch, case, break,`
- `class, public, private, protected, new, delete, template, this, virtual,`
- `try, catch, throw.`

frequently used keywords

Note: there are many more … you may see them in the examples that follow

Example 6: prefix and postfix

```
// Example 6

#include <iostream.h>

int main (){
int x =3, y=3;
cout <<++x<<endl;
cout <<y++<<endl;}
```

++ (prefix) is a **unary** operator

<< is a **binary** operator

++ (postfix) is a **unary** operator

Output:
3
4

Example 7: a C+ ternary operator

```
// Example 7

#include <iostream.h>

int main (){
int x =3, y=4;
cout <<"The max. of x and y is: "
     << (x>y?x:y);
}
```

Output:

The max. of x and y is: 4

`expression1?expression2:expression3`

Means: *if expression1 then expression2 else expression3*

Program with Three Functions

```
// Example 8
#include <iostream.h>
// declares these 2 functions
int  Square ( int );
int  Cube ( int ) ;
int  main ( void ){
cout  <<  "The square of 27 is "
<<   Square (27) << endl ;// function call
cout  <<  "The cube of 27 is "
<<  Cube (27) << endl ;// function call
     return 0 ;
}
int Square ( int  n ){return   n * n ;}
int Cube ( int  n ){return  n * n * n ;}
```

Output:

The square of 27 is 729
The cube of 27 is 19683

Precedence of Some C++ Operators

Precedence	Operator	Description
Higher	( )	Function call
	+	Positive
	-	Negative
	*	Multiplication
	/	Division
	%	Modulus (remainder)
	+	Addition
	-	Subtraction
Lower	=	Assignment

NOTE: Write some programs to test your understanding of precedence

Type Casting is Explicit Conversion of Type

```
// Example 9
#include <iostream.h>

int  main ( void ){
cout  <<  "int(4.8) ="<<int(4.8)<<endl;
cout << "float(5) ="<<float(5)<<endl;
cout <<”float(2/4)="<<float(2/4)<<endl;
cout<<"float(2)/float(4)="<<float(2)/float(4)<<endl;
}
```

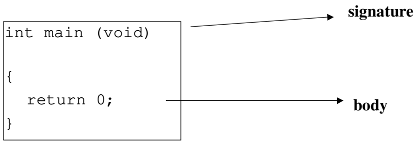
Output:

int(4.8) =4
float(5) =5
float(2/4)=0
float(2)/float(4)=0.5

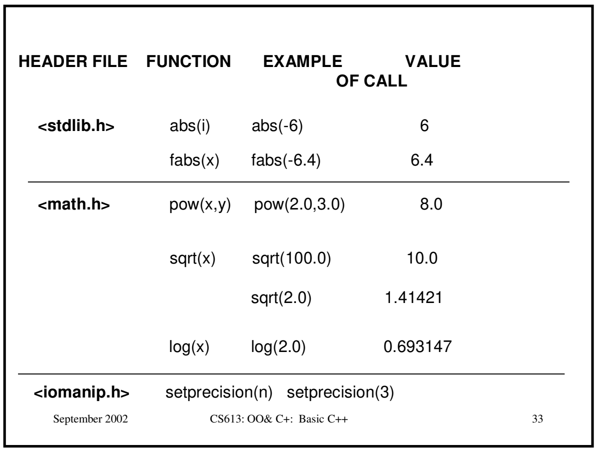
Output of `float` may look like an `int`

Parts of a Function

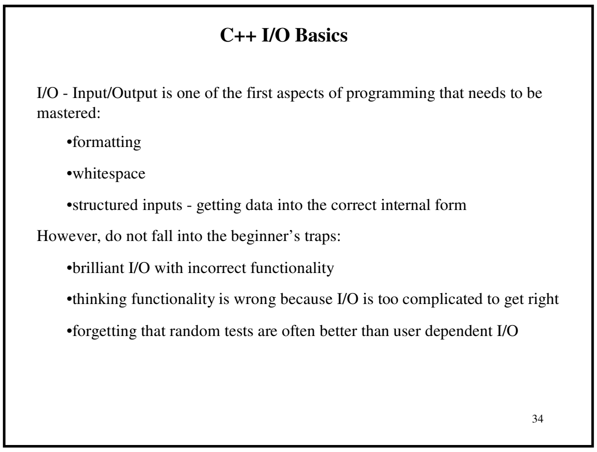
Every function has 2 parts



```
int main (void)

{
  return 0;
}
```



HEADER FILE	FUNCTION	EXAMPLE	VALUE OF CALL
<stdlib.h>	abs(i)	abs(-6)	6
	fabs(x)	fabs(-6.4)	6.4
<math.h>	pow(x,y)	pow(2.0,3.0)	8.0
	sqrt(x)	sqrt(100.0)	10.0
		sqrt(2.0)	1.41421
	log(x)	log(2.0)	0.693147
<iomanip.h>	setprecision(n)	setprecision(3)	



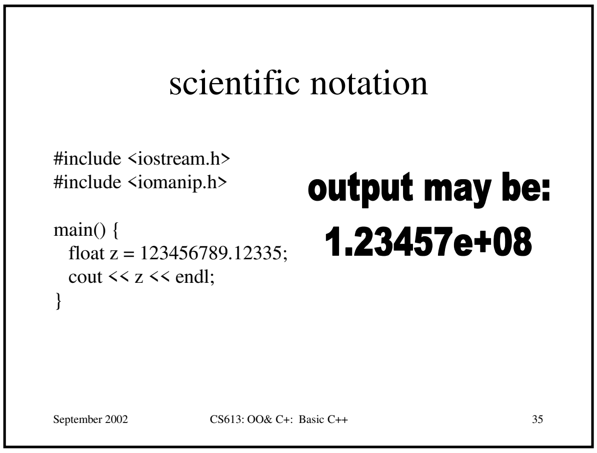
C++ I/O Basics

I/O - Input/Output is one of the first aspects of programming that needs to be mastered:

- •formatting
- •whitespace
- •structured inputs - getting data into the correct internal form

However, do not fall into the beginner’s traps:

- •brilliant I/O with incorrect functionality
- •thinking functionality is wrong because I/O is too complicated to get right
- •forgetting that random tests are often better than user dependent I/O

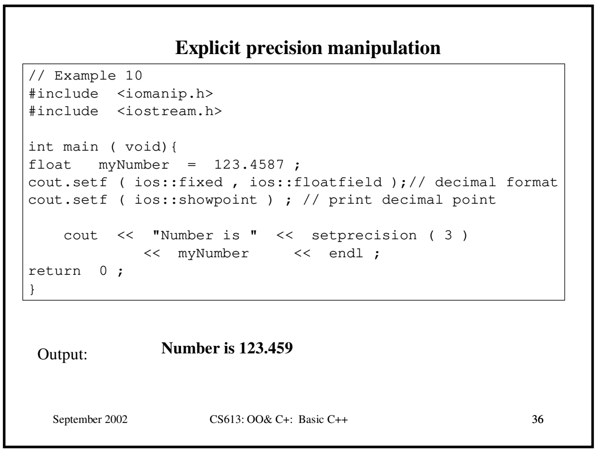


scientific notation

```
#include <iostream.h>
#include <iomanip.h>

main() {
  float z = 123456789.12335;
  cout << z << endl;
}
```

output may be:

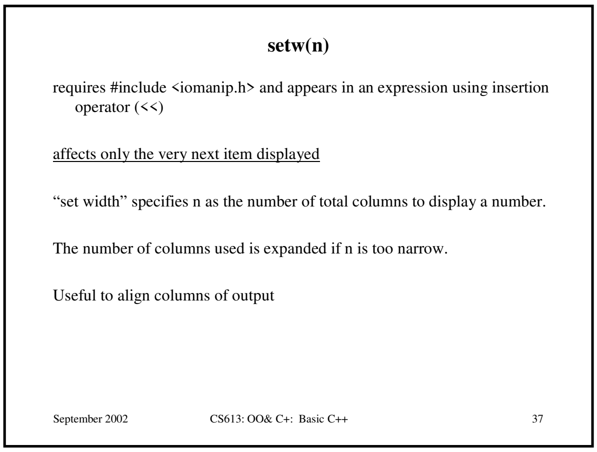
1.23457e+08



Explicit precision manipulation

```
// Example 10
#include  <iomanip.h>
#include  <iostream.h>

int main ( void){
float   myNumber  =  123.4587 ;
cout.setf ( ios::fixed , ios::floatfield );// decimal format
cout.setf ( ios::showpoint ) ; // print decimal point

    cout  <<  "Number is "  <<  setprecision ( 3 )
             <<  myNumber     <<  endl ;
return  0 ;
}
```

Output: **Number is 123.459**



setw(n)

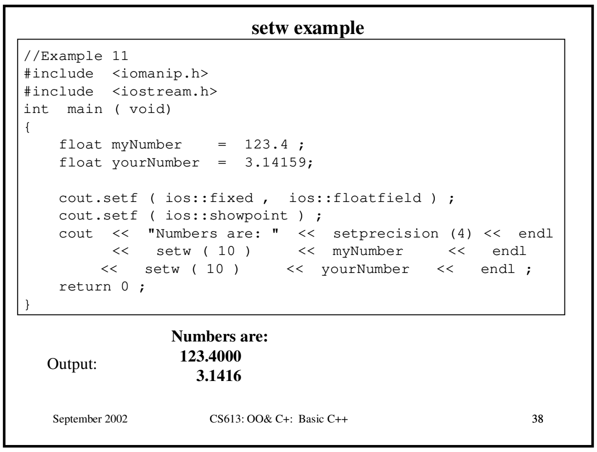
requires #include <iomanip.h> and appears in an expression using insertion operator (<<)

affects only the very next item displayed

“set width” specifies n as the number of total columns to display a number.

The number of columns used is expanded if n is too narrow.

Useful to align columns of output

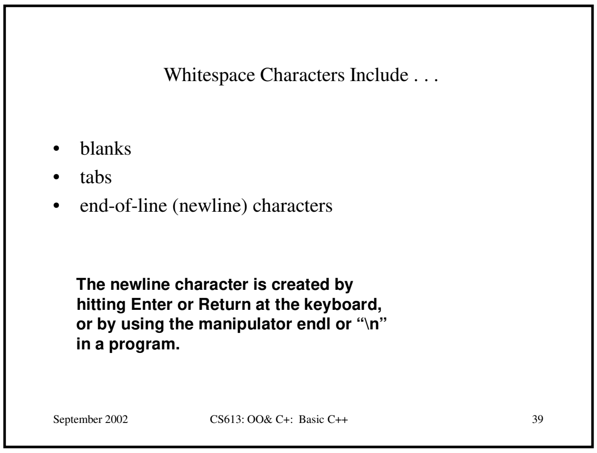


setw example

```
//Example 11
#include  <iomanip.h>
#include  <iostream.h>
int  main ( void)
{
    float myNumber    =  123.4 ;
    float yourNumber  =  3.14159;

    cout.setf ( ios::fixed ,  ios::floatfield ) ;
    cout.setf ( ios::showpoint ) ;
    cout  <<  "Numbers are: "  <<  setprecision (4) <<  endl
          <<   setw ( 10 )     <<  myNumber     <<   endl
         <<   setw ( 10 )     <<  yourNumber   <<   endl ;
    return 0 ;
}
```

Output:

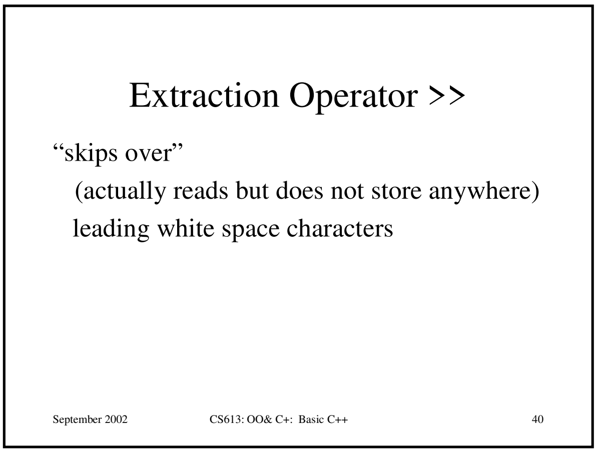
```
Numbers are:
  123.4000
    3.1416
```



Whitespace Characters Include . . .

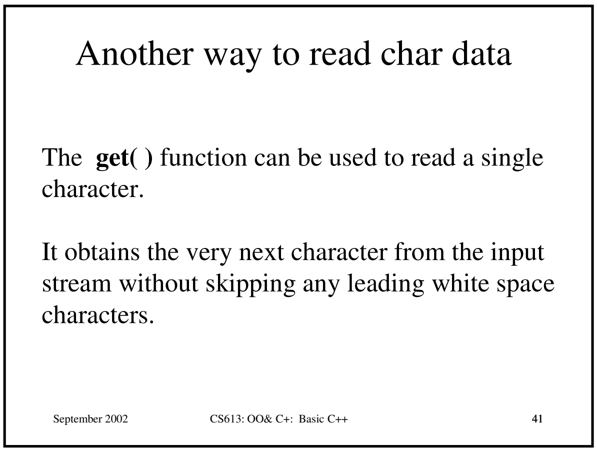
- blanks
- tabs
- end-of-line (newline) characters

The newline character is created by hitting Enter or Return at the keyboard, or by using the manipulator endl or “\n” in a program.



Extraction Operator >>

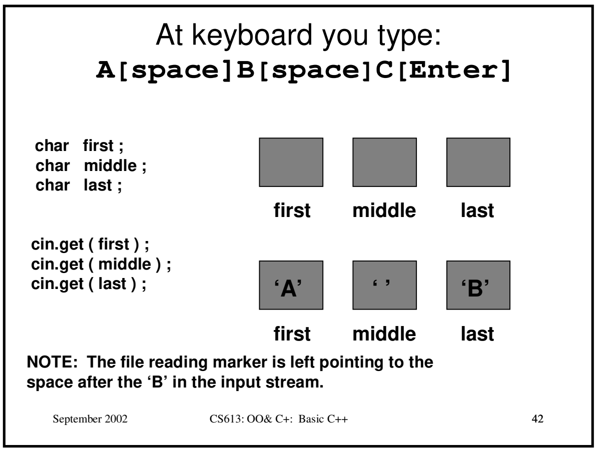
“skips over”
(actually reads but does not store anywhere)
leading white space characters



Another way to read char data

The **get( )** function can be used to read a single character.

It obtains the very next character from the input stream without skipping any leading white space characters.



At keyboard you type: `A[space]B[space]C[Enter]`

```
char   first ;
char   middle ;
char   last ;
```

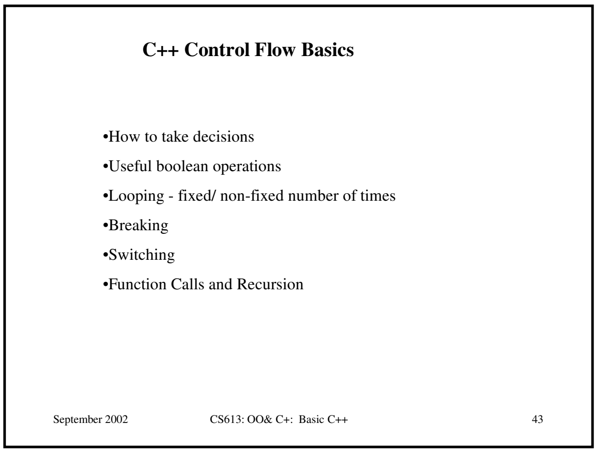
first **middle** **last**

```
cin.get ( first ) ;
cin.get ( middle ) ;
cin.get ( last ) ;
```

‘A’ ‘ ’ ‘B’

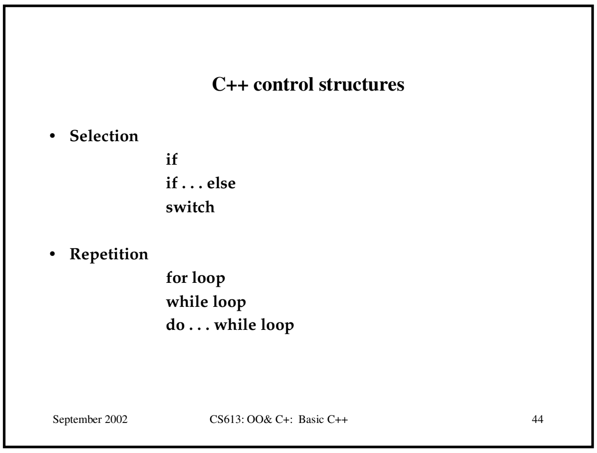
first **middle** **last**

NOTE: The file reading marker is left pointing to the space after the ‘B’ in the input stream.



C++ Control Flow Basics

- How to take decisions
- Useful boolean operations
- Looping - fixed/ non-fixed number of times
- Breaking
- Switching
- Function Calls and Recursion



C++ control structures

- **Selection**
  - **if**
  - **if . . . else**
  - **switch**
- **Repetition**
  - **for loop**
  - **while loop**
  - **do . . . while loop**

CONTROL STRUCTURES

Use logical expressions which may include:

6 Relational Operators

<   <=   >   >=   ==   !=

3 Logical Operators

!   && ||

Operator	Meaning	Associativity
!	NOT	Right
*, / , %	Multiplication, Division, Modulus	Left
+ , -	Addition, Subtraction	Left
<	Less than	Left
<=	Less than or equal to	Left
>	Greater than	Left
>=	Greater than or equal to	Left
==	Is equal to	Left
!=	Is not equal to	Left
&&	AND	Left
| |	OR	Left
=	Assignment	Right

“SHORT-CIRCUIT” EVALUATION

- **C++ uses short circuit evaluation of logical expressions**
- **this means that evaluation stops as soon as the final truth value can be determined**

Short-Circuit Example

int Age, Height;

Age = 25;
Height = 70;

EXPRESSION

$\underbrace{\textbf{(Age > 50)}}_{\textbf{false}}$ **&& (Height > 60)**

Evaluation can stop now

Better example

int Number;
float X;

(Number != 0) && (X < 1 / Number)

Compound statement

- We use braces { } to build compound -statements
- To use braces or not to use braces??

```
for (i = 0; i < n; ++i)
     sum += i;
```

```
for (i = 0; i < n; ++i) {
     sum += i;
}
```

Conditional statements

- Syntax

```
if (expression)
   statement1
else
   statement2
```

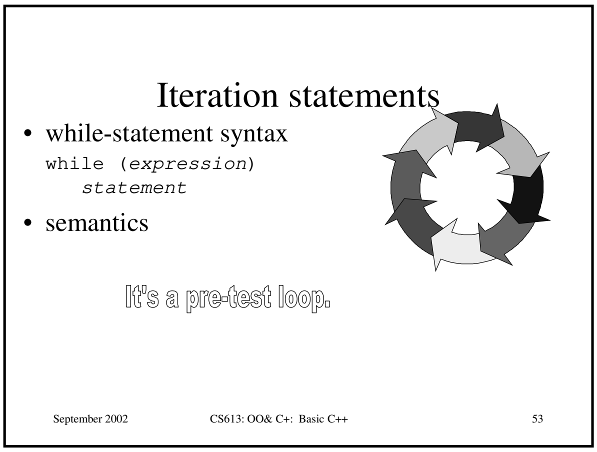
else clause is optional

Beware of *dangling else*

what's the output?

end of output

```
//Example 12
#include  <iostream.h>
int  main ( void) {
int x = 7, y = 8;
if (x == 0)
   if (y == 0) cout << "yes"<< endl;
else   cout << "no"<<  endl;
cout << "end of output" << endl;
}
```



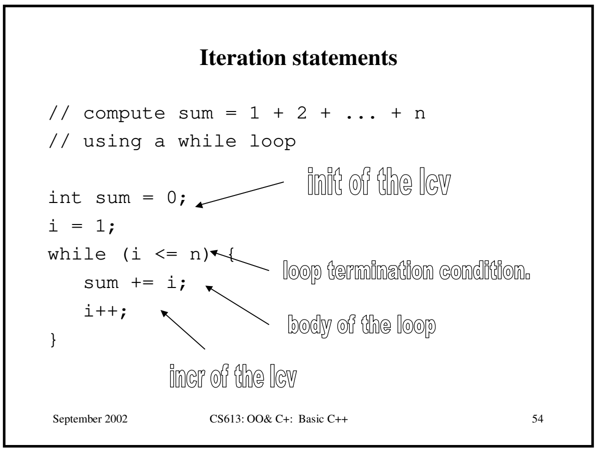
Iteration statements

- while-statement syntax

```
while (expression)
   statement
```

- semantics

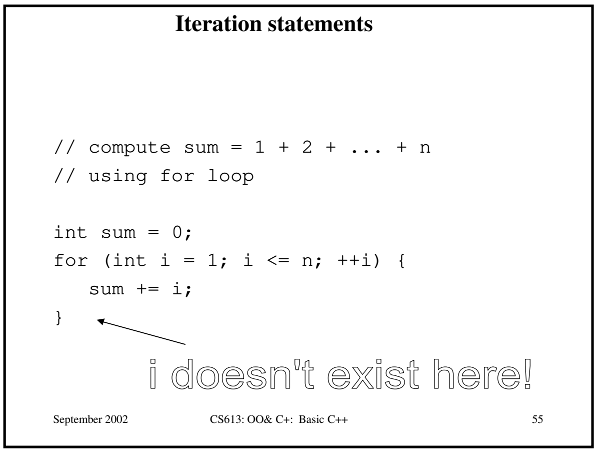
It's a pre-test loop.



Iteration statements

```
// compute sum = 1 + 2 + ... + n
// using a while loop

int sum = 0;
i = 1;
while (i <= n) {
   sum += i;
   i++;
}
```

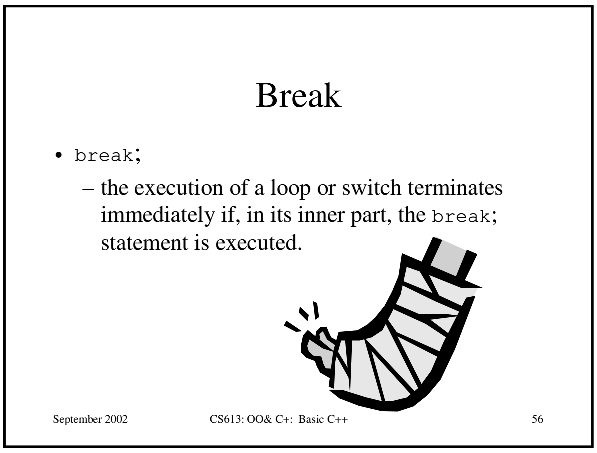
init of the lcv
loop termination condition.
body of the loop
incr of the lcv



Iteration statements

```
// compute sum = 1 + 2 + ... + n
// using for loop

int sum = 0;
for (int i = 1; i <= n; ++i) {
   sum += i;
}
```

i doesn't exist here!



Break

- `break`;
  - the execution of a loop or switch terminates immediately if, in its inner part, the `break`; statement is executed.

Combining *break* and *for*

```
char ch;
int count = 0;

for ( ;   ; )   {
   cin >> ch;
   if (ch == ‘\n’) break;
   ++count;
}
```

Who would do this?

Switch

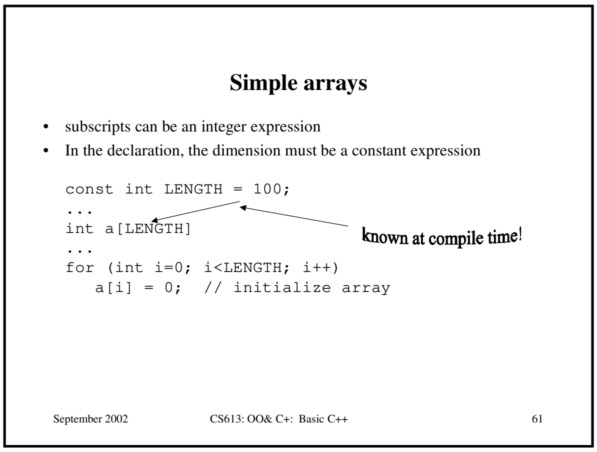
```
switch (letter) {
   case ‘N’: cout < “New York\n”;
             break;
   case ‘L’: cout < “London\n”;
             break;
   case ‘A’: cout < “Amsterdam\n”;
             break;
   default:  cout < “Somewhere else\n”;
             break;
}
```

Switch

```
switch (letter) {
   case ‘N’: case ‘n’: cout < “New York\n”;
                       break;
   case ‘L’: case ‘l’: cout < “London\n”;
                       break;
   case ‘A’: case ‘a’: cout < “Amsterdam\n”;
                       break;
   default:  cout < “Somewhere else\n”;
             break;
}
```

What’s the output?

```
int i = 0;
switch (  i ) {
   case 0 :   i+= 5;
   case 1 :   ++i;
   case 2 :   ++i;
   case 3 :   ++i;
   default    ++i;
}
cout << “i is: “ << i << endl;
```

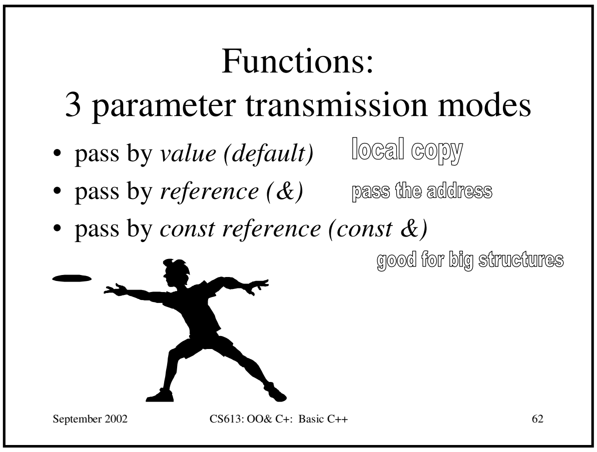


Simple arrays

- subscripts can be an integer expression
- In the declaration, the dimension must be a constant expression

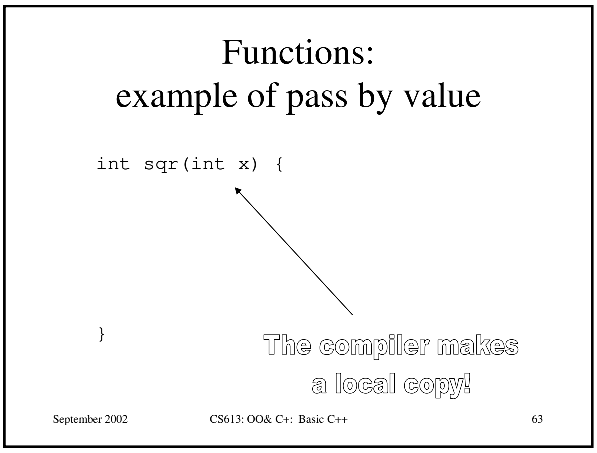
```
const int LENGTH = 100;
...
int a[LENGTH]
...
for (int i=0; i<LENGTH; i++)
   a[i] = 0;  // initialize array
```

known at compile time!



Functions: 3 parameter transmission modes

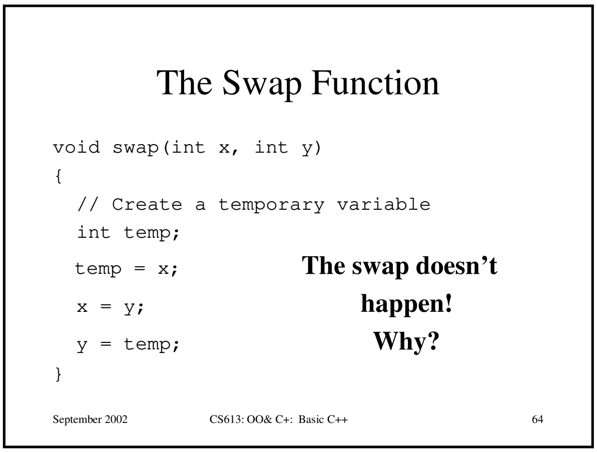
- pass by *value (default)* local copy
- pass by *reference (&)* pass the address
- pass by *const reference (const &)* good for big structures



Functions: example of pass by value

```
int sqr(int x) {



}
```

The compiler makes a local copy!



The Swap Function

```
void swap(int x, int y)
{
  // Create a temporary variable
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

The swap doesn't happen! Why?

Passing values by reference

- C/C++ passes parameters by value, i.e. a copy of the variable is passed to the function, not the actual value itself.
- C++ can pass the actual variables themselves - known as *passing parameters by reference*.
- To pass a parameter by reference we place & between the parameters type name and the parameter tag.

The New Swap Function

```
void swap(int& x, int& y)
{
  // Create a temporary variable
  int temp;

  temp = x;
  x = y;
  y = temp;
}
```

What about functions and arrays / structures?

Functions:
example of pass by reference

```
void swap(int & x, int & y) {
```

Address

Must pass by reference!

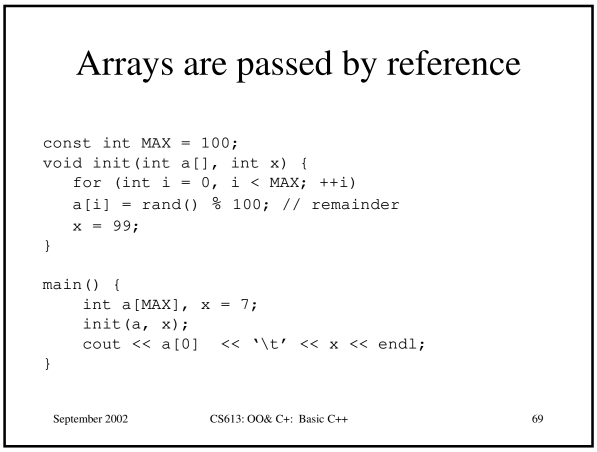
```
}
```

Functions: pass by
const reference

- Makes sense with large structures or objects

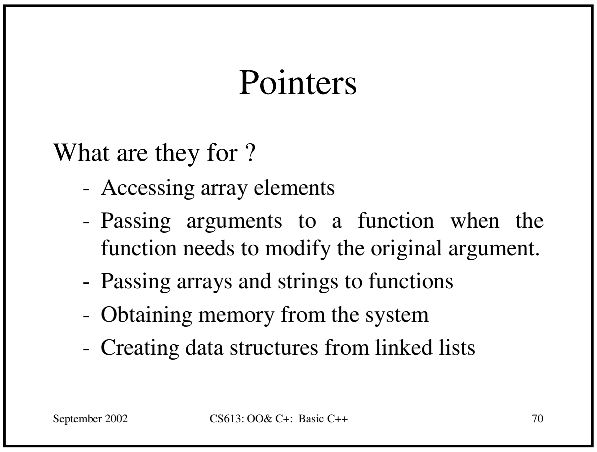
**We'll use it when
we make objects.**

- const &



Arrays are passed by reference

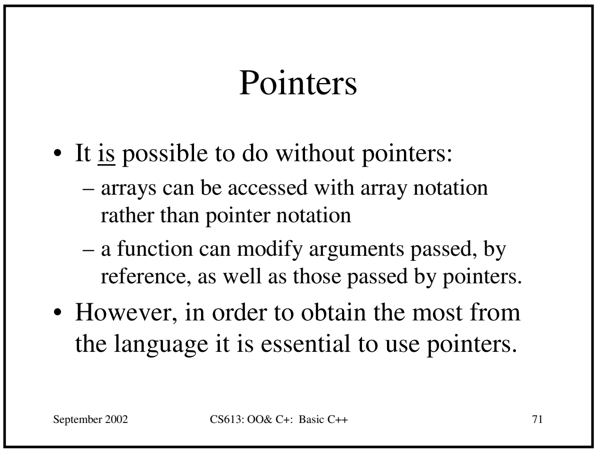
```
const int MAX = 100;
void init(int a[], int x) {
   for (int i = 0, i < MAX; ++i)
   a[i] = rand() % 100; // remainder
   x = 99;
}

main() {
    int a[MAX], x = 7;
    init(a, x);
    cout << a[0]  << ‘\t’ << x << endl;
}
```



Pointers

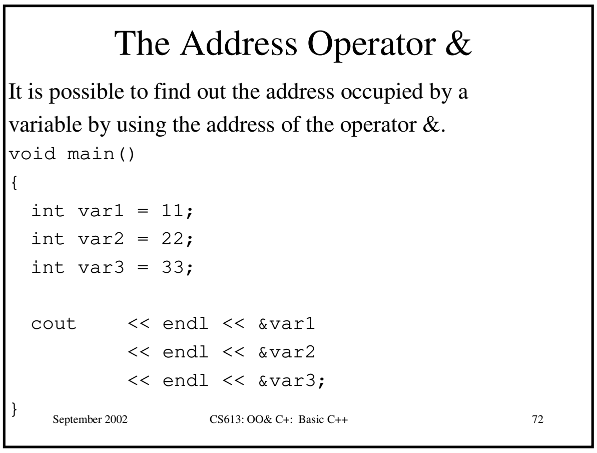
What are they for ?

- Accessing array elements
- Passing arguments to a function when the function needs to modify the original argument.
- Passing arrays and strings to functions
- Obtaining memory from the system
- Creating data structures from linked lists



Pointers

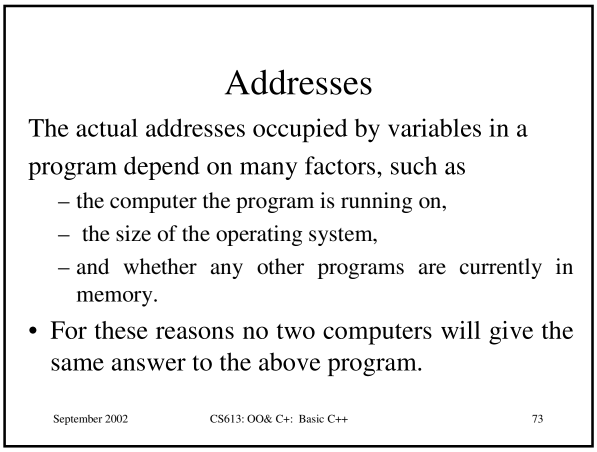
- It is possible to do without pointers:
  - arrays can be accessed with array notation rather than pointer notation
  - a function can modify arguments passed, by reference, as well as those passed by pointers.
- However, in order to obtain the most from the language it is essential to use pointers.



The Address Operator &

It is possible to find out the address occupied by a variable by using the address of the operator &.

```
void main()
{
  int var1 = 11;
  int var2 = 22;
  int var3 = 33;

  cout    << endl << &var1
          << endl << &var2
          << endl << &var3;
}
```

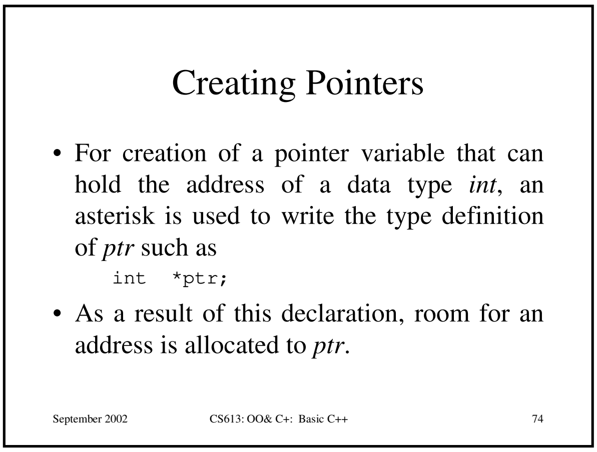


Addresses

The actual addresses occupied by variables in a program depend on many factors, such as

- the computer the program is running on,
- the size of the operating system,
- and whether any other programs are currently in memory.

- For these reasons no two computers will give the same answer to the above program.

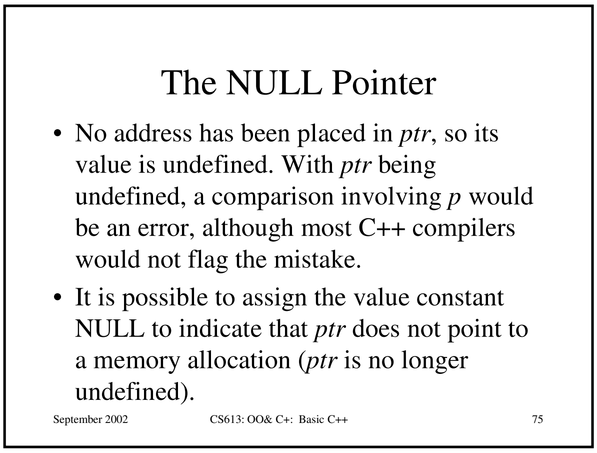


Creating Pointers

- For creation of a pointer variable that can hold the address of a data type *int*, an asterisk is used to write the type definition of *ptr* such as

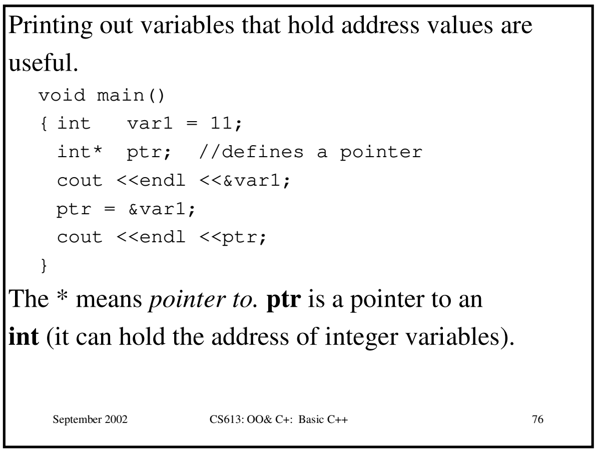
```
int  *ptr;
```

- As a result of this declaration, room for an address is allocated to *ptr*.



The NULL Pointer

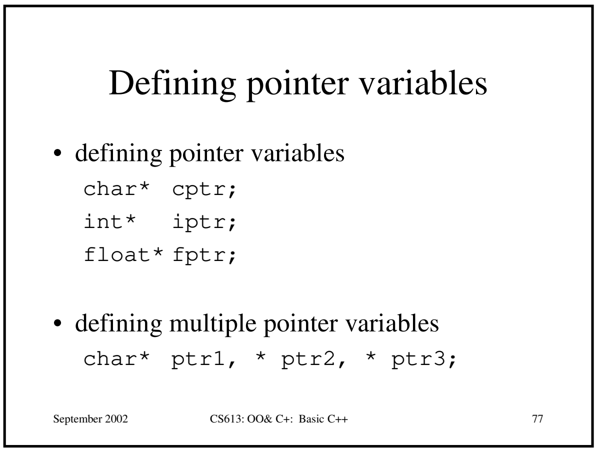
- No address has been placed in *ptr*, so its value is undefined. With *ptr* being undefined, a comparison involving *p* would be an error, although most C++ compilers would not flag the mistake.
- It is possible to assign the value constant NULL to indicate that *ptr* does not point to a memory allocation (*ptr* is no longer undefined).



Printing out variables that hold address values are useful.

```
void main()
{ int   var1 = 11;
  int*  ptr;  //defines a pointer
  cout <<endl <<&var1;
  ptr = &var1;
  cout <<endl <<ptr;
}
```

The * means *pointer to.* **ptr** is a pointer to an **int** (it can hold the address of integer variables).



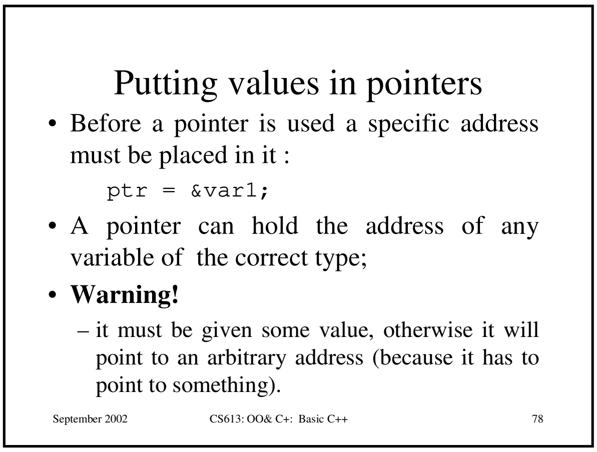
Defining pointer variables

- defining pointer variables

```
char*  cptr;
int*   iptr;
float* fptr;
```

- defining multiple pointer variables

```
char*  ptr1, * ptr2, * ptr3;
```

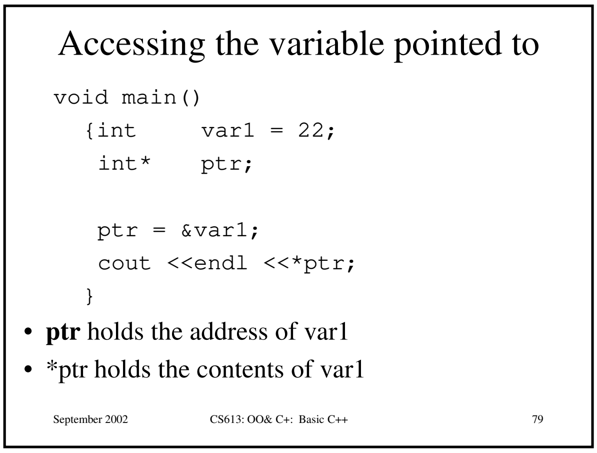


Putting values in pointers

- Before a pointer is used a specific address must be placed in it :

```
   ptr = &var1;
```

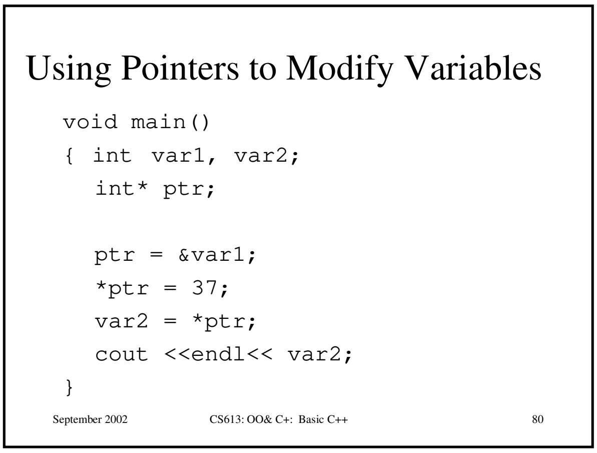
- A pointer can hold the address of any variable of the correct type;
- **Warning!**
  - – it must be given some value, otherwise it will point to an arbitrary address (because it has to point to something).



Accessing the variable pointed to

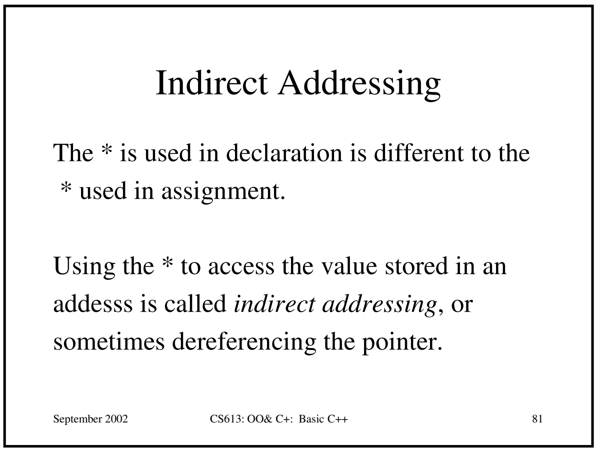
```
void main()
  {int     var1 = 22;
   int*    ptr;

   ptr = &var1;
   cout <<endl <<*ptr;
  }
```

- **ptr** holds the address of var1
- *ptr holds the contents of var1



Using Pointers to Modify Variables

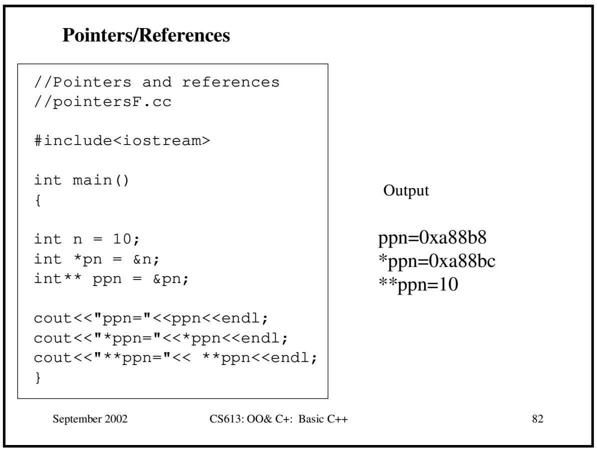
```
void main()
{  int  var1, var2;
   int* ptr;

   ptr = &var1;
   *ptr = 37;
   var2 = *ptr;
   cout <<endl<< var2;
}
```



Indirect Addressing

The * is used in declaration is different to the * used in assignment.

Using the * to access the value stored in an addesss is called *indirect addressing*, or sometimes dereferencing the pointer.

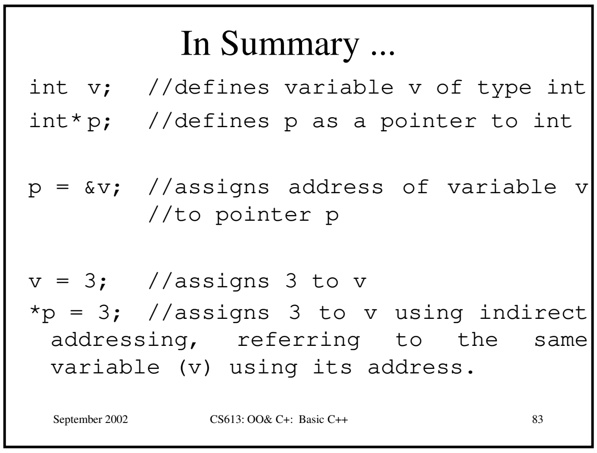


Pointers/References

```
//Pointers and references
//pointersF.cc

#include<iostream>

int main()
{

int n = 10;
int *pn = &n;
int** ppn = &pn;

cout<<"ppn="<<ppn<<endl;
cout<<"*ppn="<<*ppn<<endl;
cout<<"**ppn="<< **ppn<<endl;
}
```

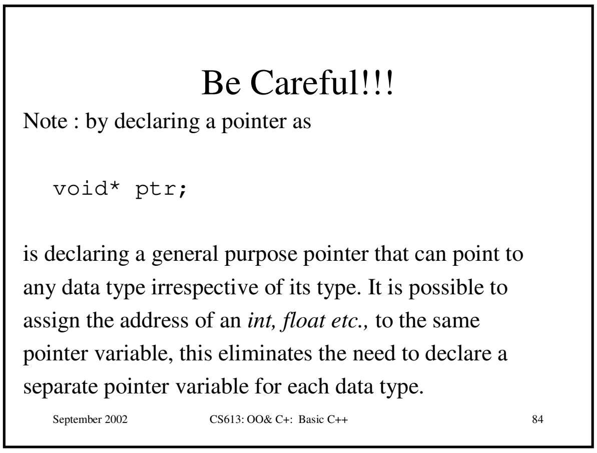
Output

```
ppn=0xa88b8
*ppn=0xa88bc
**ppn=10
```



In Summary ...

```
int  v;   //defines variable v of type int
int* p;   //defines p as a pointer to int

p = &v;  //assigns address of variable v
         //to pointer p

v = 3;   //assigns 3 to v
*p = 3;  //assigns 3 to v using indirect
  addressing,  referring  to  the  same
  variable (v) using its address.
```

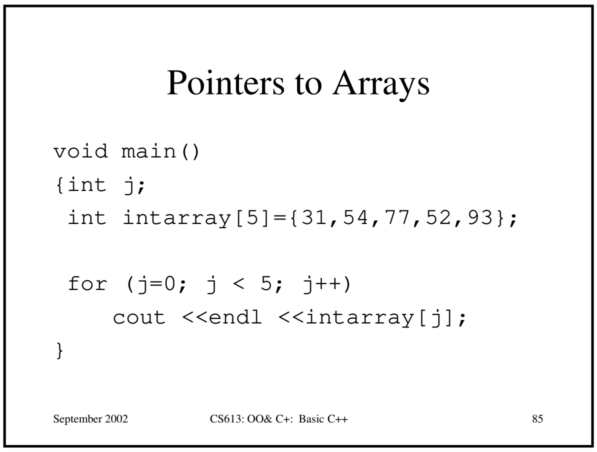


Be Careful!!!

Note : by declaring a pointer as

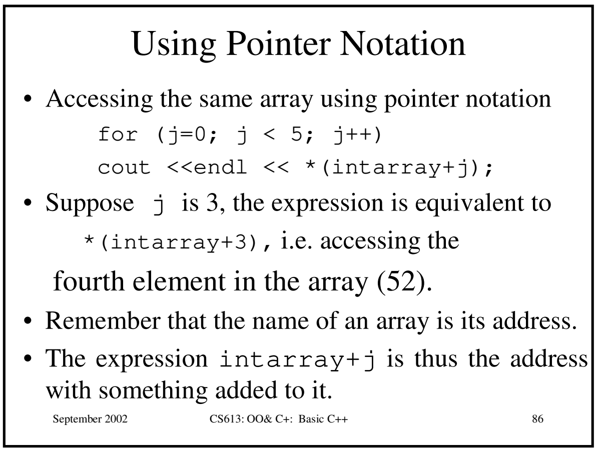
```
  void* ptr;
```

is declaring a general purpose pointer that can point to any data type irrespective of its type. It is possible to assign the address of an *int, float etc.,* to the same pointer variable, this eliminates the need to declare a separate pointer variable for each data type.



Pointers to Arrays

```
void main()
{int j;
 int intarray[5]={31,54,77,52,93};

 for (j=0; j < 5; j++)
     cout <<endl <<intarray[j];
}
```

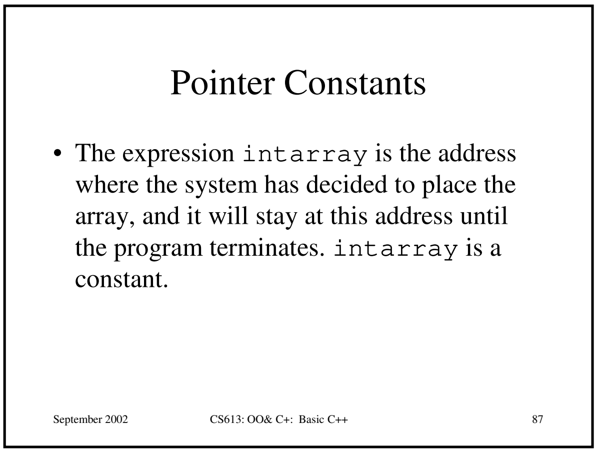


Using Pointer Notation

- Accessing the same array using pointer notation

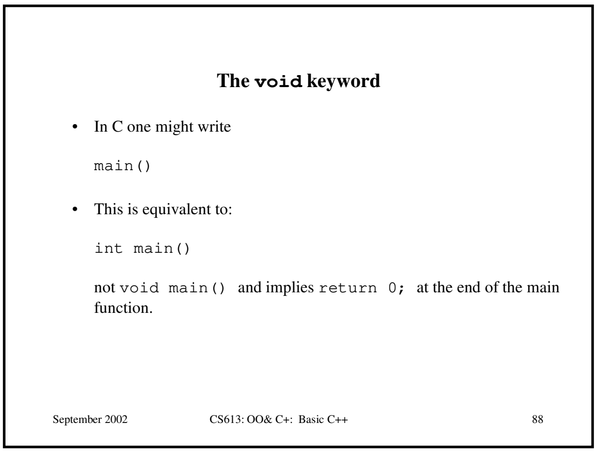
```
    for (j=0; j < 5; j++)
    cout <<endl << *(intarray+j);
```

- Suppose `j` is 3, the expression is equivalent to `*(intarray+3)`, i.e. accessing the fourth element in the array (52).
- Remember that the name of an array is its address.
- The expression `intarray+j` is thus the address with something added to it.



Pointer Constants

- The expression `intarray` is the address where the system has decided to place the array, and it will stay at this address until the program terminates. `intarray` is a constant.



The `void` keyword

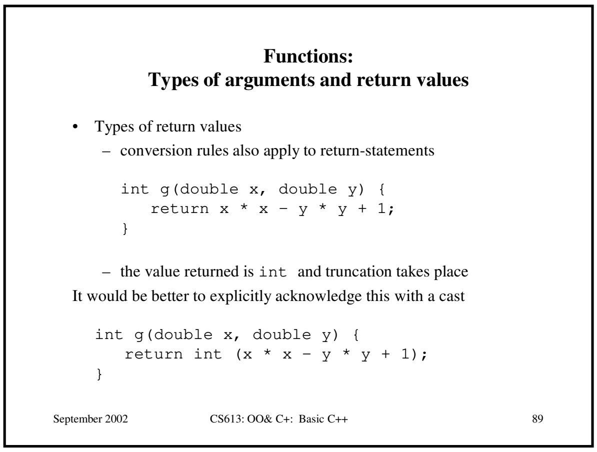
- In C one might write

  `main()`

- This is equivalent to:

  `int main()`

  not `void main()` and implies `return 0;` at the end of the main function.



Functions:
Types of arguments and return values

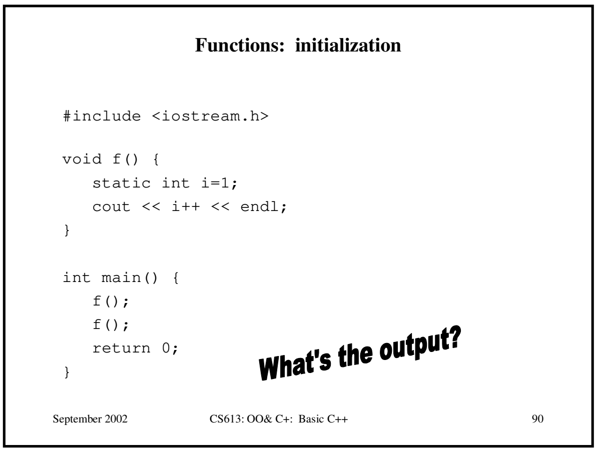
- Types of return values
  - conversion rules also apply to return-statements

```
int g(double x, double y) {
   return x * x - y * y + 1;
}
```

  - the value returned is `int` and truncation takes place

It would be better to explicitly acknowledge this with a cast

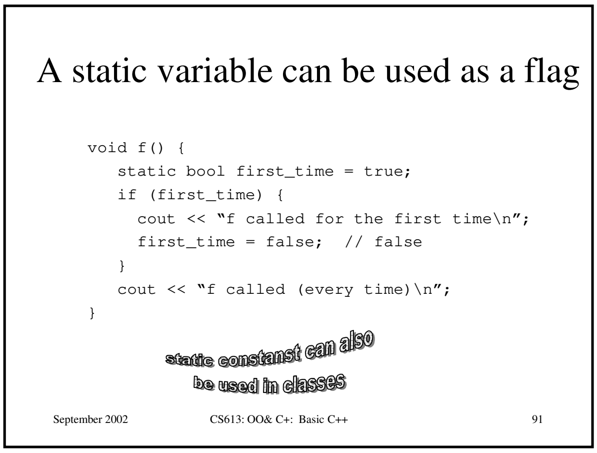
```
int g(double x, double y) {
   return int (x * x - y * y + 1);
}
```



Functions: initialization

```
#include <iostream.h>

void f() {
   static int i=1;
   cout << i++ << endl;
}

int main() {
   f();
   f();
   return 0;
}
```

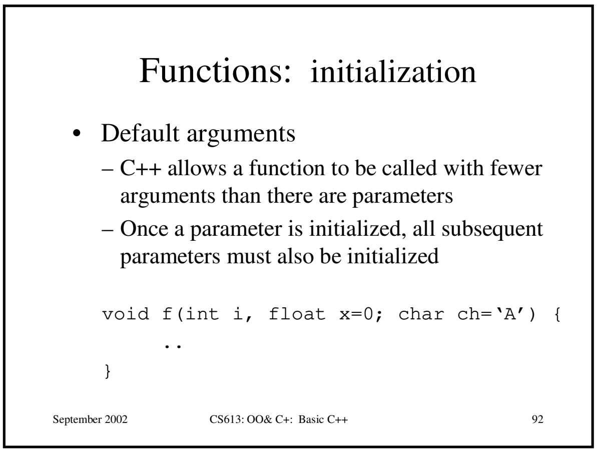
What's the output?



A static variable can be used as a flag

```
void f() {
   static bool first_time = true;
   if (first_time) {
     cout << “f called for the first time\n”;
     first_time = false;  // false
   }
   cout << “f called (every time)\n”;
}
```

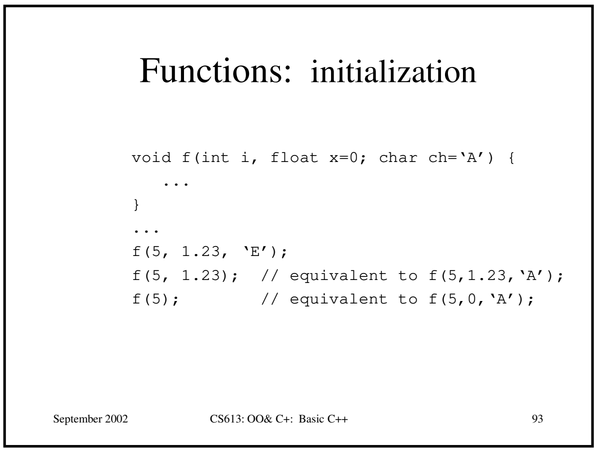
static constanst can also
be used in classes



Functions: initialization

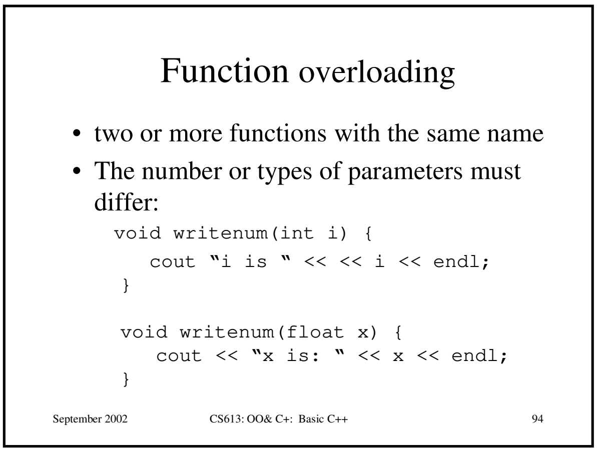
- Default arguments
  - C++ allows a function to be called with fewer arguments than there are parameters
  - Once a parameter is initialized, all subsequent parameters must also be initialized

```
void f(int i, float x=0; char ch=‘A’) {
      ..
}
```



Functions: initialization

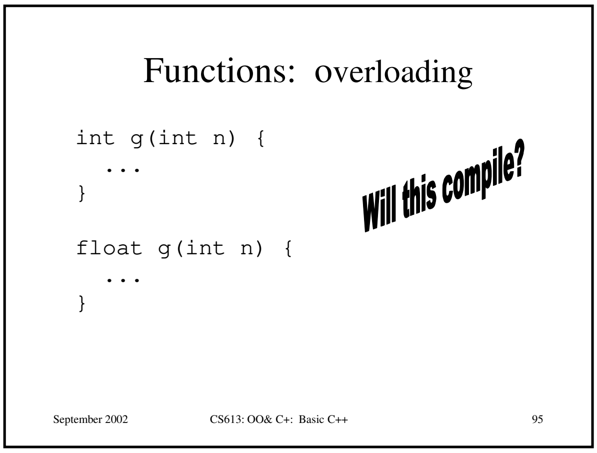
```
void f(int i, float x=0; char ch=‘A’) {
   ...
}
...
f(5, 1.23, ‘E’);
f(5, 1.23);  // equivalent to f(5,1.23,‘A’);
f(5);        // equivalent to f(5,0,‘A’);
```



Function overloading

- two or more functions with the same name
- The number or types of parameters must differ:

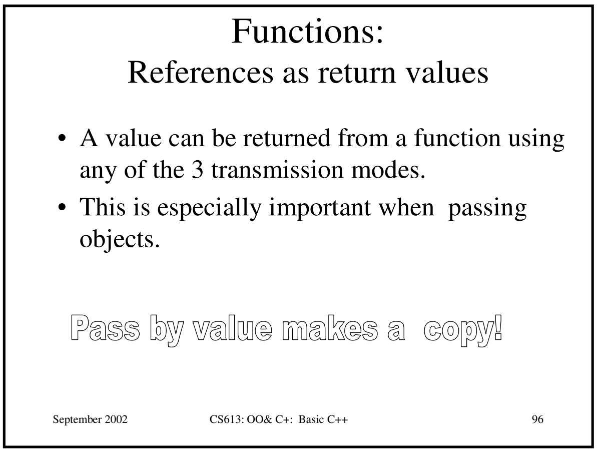
```
void writenum(int i) {
   cout “i is “ << << i << endl;
 }

 void writenum(float x) {
    cout << “x is: “ << x << endl;
 }
```



Functions: overloading

```
int g(int n) {
  ...
}

float g(int n) {
  ...
}
```

Will this compile?



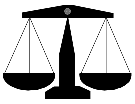
Functions:
References as return values

- A value can be returned from a function using any of the 3 transmission modes.
- This is especially important when passing objects.

Pass by value makes a copy!

Functions:
Inline functions and macros

- A function call causes
  - a jump to a separate and unique code segment
  - the passing and returning of arguments and function values
  - saving the state
- Inline functions cause
  - no jump or parameter passing
  - no state saving
  - duplication of the code segment in place of the function call



What's better: #define, or const

- #define is a preprocessor directive
  - thus, constants so defined are not seen by the compiler, or the debugger.
  - might get confusing error message about a number where you used a constant
- soln: use const
  - but HOW?

Warning in advance -constants in a class:

```
class Game {
private:
   static const int NUM_TURNS = 5;
   int scores[NUM_TURNS];
};


const int Game::NUM_TURNS;
```

declaration

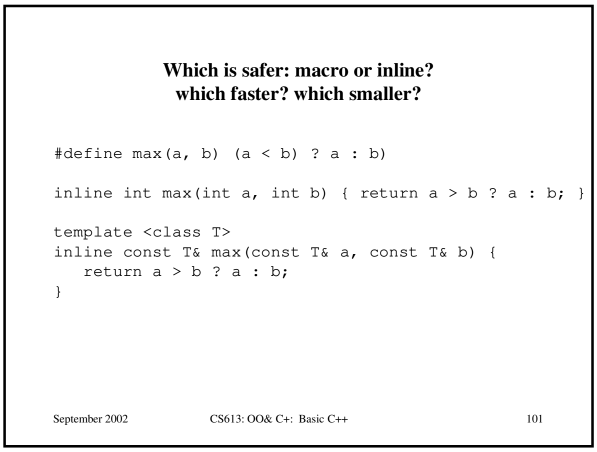
definition

this is ansi compliant, but some compilers might not have caught up. gcc has!

constants in a class (old style):

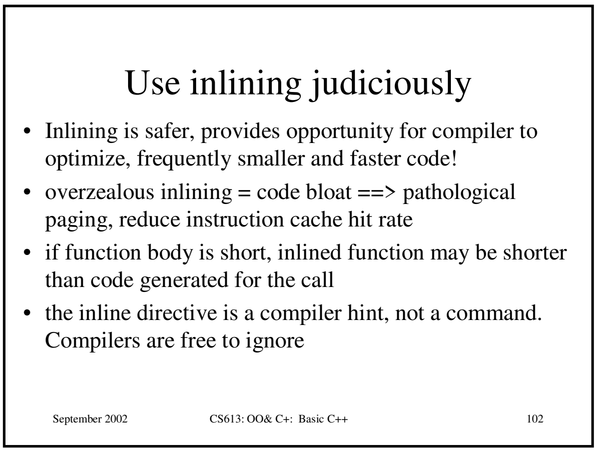
```
class Game {
private:
  enum { NUM_TURNS = 5 };
  int scores[NUM_TURNS];
};
```

affectionately known as the "enum hack"! But don't have to do it this way anymore.



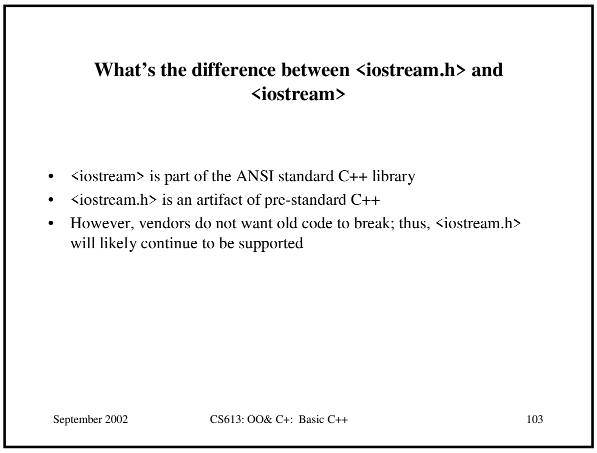
Which is safer: macro or inline? which faster? which smaller?

```
#define max(a, b) (a < b) ? a : b)

inline int max(int a, int b) { return a > b ? a : b; }

template <class T>
inline const T& max(const T& a, const T& b) {
   return a > b ? a : b;
}
```



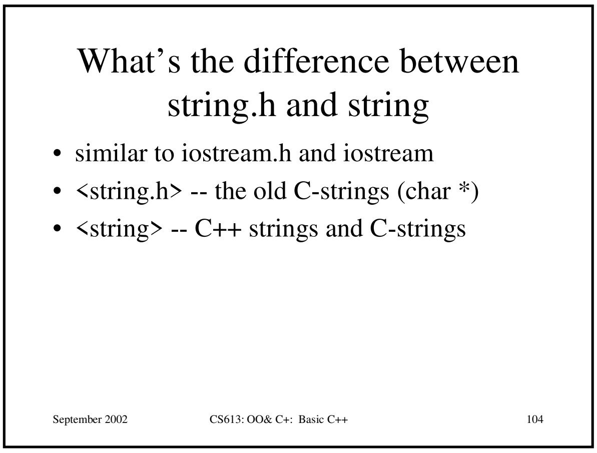
Use inlining judiciously

- Inlining is safer, provides opportunity for compiler to optimize, frequently smaller and faster code!
- overzealous inlining = code bloat ==> pathological paging, reduce instruction cache hit rate
- if function body is short, inlined function may be shorter than code generated for the call
- the inline directive is a compiler hint, not a command. Compilers are free to ignore



What's the difference between <iostream.h> and <iostream>

- <iostream> is part of the ANSI standard C++ library
- <iostream.h> is an artifact of pre-standard C++
- However, vendors do not want old code to break; thus, <iostream.h> will likely continue to be supported



What's the difference between string.h and string

- similar to iostream.h and iostream
- <string.h> -- the old C-strings (char *)
- <string> -- C++ strings and C-strings

What’s the algorithm for old/new

- old headers are likely to continue to be supported, even though they’re not in the ANSI standard
- new C++ headers have same name w/out the .h, but contents are in std
- C headers, like <stdio.h> continue, but not in std
- new C++ headers for functionality in old C library have names like <cstdio>; offer same content but are in std

Preprocessor facilities

- Conditional compilation
  - a useful way to handle multiple include files

```
#ifndef SOME___HEADER___FILE
     #include “SOME___HEADER___FILE”
#endif
```

System files automatically do this:
for example <iostream.h>

C strings:

- C++ has them but: artifact of C
- *programmer must manage memory*
- terminated with *‘\0’*
- built-in c-string library:
  - strlen(s): returns length of string
  - strcat(s, t): place *t* at end of *s*
  - strcpy(s, t): copy *t* into buffer *s*
  - strcmp(s, t): 0 if s==t, <0 if s<t, >0 if s>t

Strings and pointers

```
//pointersA.cc

#include<iostream>
#include<cstring>

int length(char *s){
if (s[0] == '\0') {return 0;}
 else{s=&s[1]; return
1+length(s);}}

int length2(char *s){
if (s[0] == '\0') {return 0;}
 else{s++; return 1+length(s);}}

char head(char *s) { return *s;}
char* tail (char*s){return ++s;}
```

```
int main()
{
char* s = ”rose";
cout << s;
cout<<"\nlength = "<<strlen(s);
cout<<"\nlength = "<<length(s);
cout<<"\nlength = "<<length2(s);
cout <<"\nhead of "<<s<<" is "<<head(s);
cout <<"\ntail  of "<<s<<" is "<<tail(s);
}
```

output

```
rose
length = 4
length = 4
length = 4
head of rose is r
tail  of rose is ose
```

Command line parameters

main(int argc, char * argv[])

argc is the number of parameters

argv is an array of char* with

argv[0] the first parameter

argv[1] the second paramter

etc.

Note that there is always at least one parameter

Command line parameters

Integer parameters must be converted:

```
main(int argc, char * argv[]) {
   if (argc < 2) {
      cout << “usage: “ << argv[0]
              << “ <number>” << endl;
      return 1;
   }
   int n = atoi(argv[1]);
```

Reading from files

- Files represent a permanent medium
- must be opened
- inside programs, files have a logical name that must be mapped to the physical name on the disk
- after reading from or writing to a file, it should be closed

Statements for using Disk I/O

```
#include <fstream>

ifstream   myInfile;                      // declarations
ofstream  myOutfile;

myInfile.open(“A:\\myIn.dat”);      // open files
myOutfile.open(“A:\\myOut.dat”);
// … do your thing here
myInfile.close( );                        // close files
myOutfile.close( );
```

What does opening a file do?

- associates the C++ identifier for your file with the physical (disk) name for the file
- if the input file does not exist on disk, open is not successful
- if the output file does not exist on disk, a new file with that name is created
- if the output file already exists, it is erased
- places a *file reading marker* at the very beginning of the file, pointing to the first character in it

```
#include <fstream>
main (int argc, char * argv[]) {
   fstream input;
   input.open(argv[1], ios::in);
   int count = 0;
   char ch;
   input.get(ch);
   while ( ! input.eof() ) {
        if (ch == ‘\n’) ++count;
        input.get(ch);
   }
   cout << “there are: ” << count
   << “lines in ” << argv[1] << endl;
}
```

This is bad programming style

There are no checks, see
- if user input the filename,
- does the file exist,
- can it be opened!

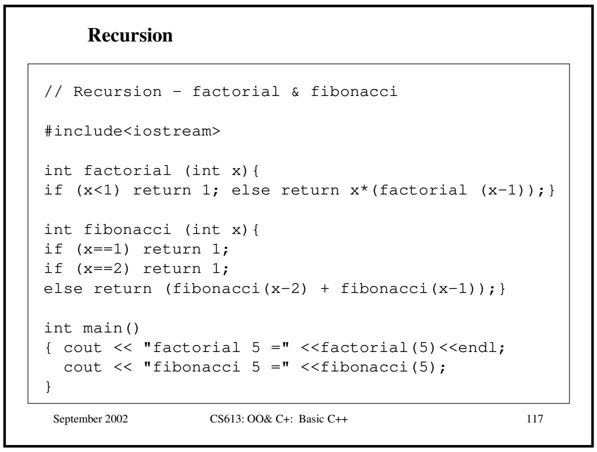
```
#include <fstream>
main (int argc, char * argv[]) {
  if (argc != 2) {
    cout << “usage: “ << argv[0] << “ <filename>” << endl;
    return 1;
  }
  fstream input;
  input.open(argv[1], ios::in);
  if (!input) {
     cout << “file: “ << argv[1] << “ does not exist!” << endl;
     return 1;
  }
  int count = 0;   char ch;   input.get(ch);
  while ( ! input.eof() ) {
     if (ch == ‘\n’) ++count;       input.get(ch);
  }
  cout << “there are: “ << count  << “  lines in “ << argv[1] << endl;
}
```

Check argument count

Make sure the file exists

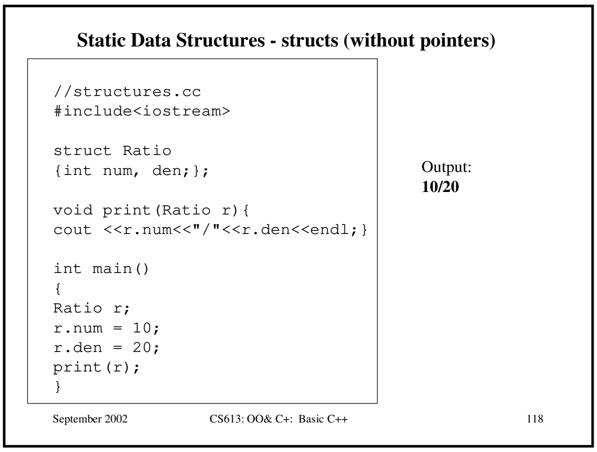
look for end of file

```
#include <fstream>
void double_space(istream & f, ostream & t) {
  char ch;
  while (  f.get(ch)  ) {
      t.put(ch);
      if (ch == ‘\n’)   t.put(ch);
  }
}
int main(int argc, char * argv[]) {
  if (argc != 3) {
      cout << “usage: “ << argv[0] << “<infile>  <outfile>”;
       return 1;
  }
   istream fin(argv[1]);    ostream fout(argv[2]);
   if (!fin) { cout << “can’t open “ << argv[1] << endl;  return 1; }
   if (!fout) { cout << “can’t open “ << argv[2] << endl;  return 1;}
   double_space(fin, fout);    fin.close();  fout.close();  return 0;
}
```



Recursion

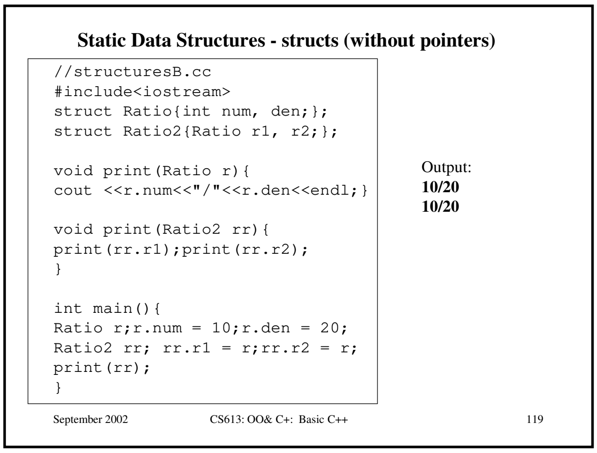
```
// Recursion - factorial & fibonacci

#include<iostream>

int factorial (int x){
if (x<1) return 1; else return x*(factorial (x-1));}

int fibonacci (int x){
if (x==1) return 1;
if (x==2) return 1;
else return (fibonacci(x-2) + fibonacci(x-1));}

int main()
{ cout << "factorial 5 =" <<factorial(5)<<endl;
  cout << "fibonacci 5 =" <<fibonacci(5);
}
```



Static Data Structures - structs (without pointers)

```
//structures.cc
#include<iostream>

struct Ratio
{int num, den;};

void print(Ratio r){
cout <<r.num<<"/"<<r.den<<endl;}

int main()
{
Ratio r;
r.num = 10;
r.den = 20;
print(r);
}
```

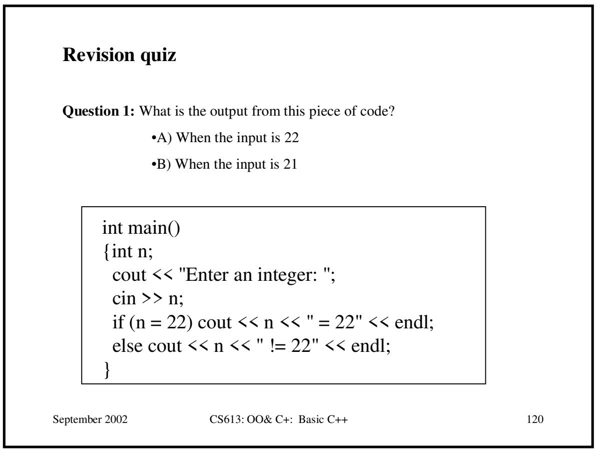
Output:
10/20



Static Data Structures - structs (without pointers)

```
//structuresB.cc
#include<iostream>
struct Ratio{int num, den;};
struct Ratio2{Ratio r1, r2;};

void print(Ratio r){
cout <<r.num<<"/"<<r.den<<endl;}

void print(Ratio2 rr){
print(rr.r1);print(rr.r2);
}

int main(){
Ratio r;r.num = 10;r.den = 20;
Ratio2 rr; rr.r1 = r;rr.r2 = r;
print(rr);
}
```

Output:
10/20
10/20

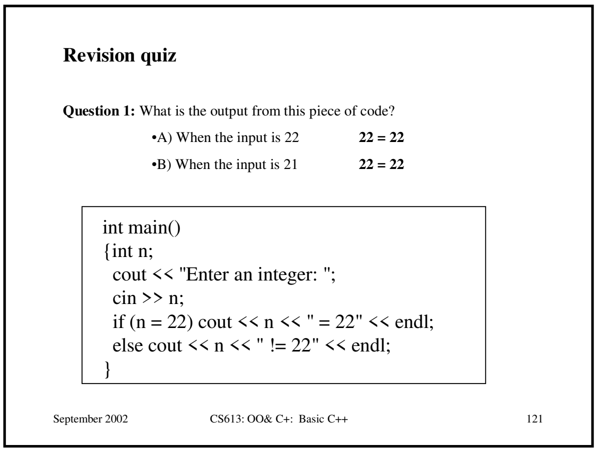


Revision quiz

Question 1: What is the output from this piece of code?

- A) When the input is 22
- B) When the input is 21

```
int main()
{int n;
  cout << "Enter an integer: ";
  cin >> n;
  if (n = 22) cout << n << " = 22" << endl;
  else cout << n << " != 22" << endl;
}
```

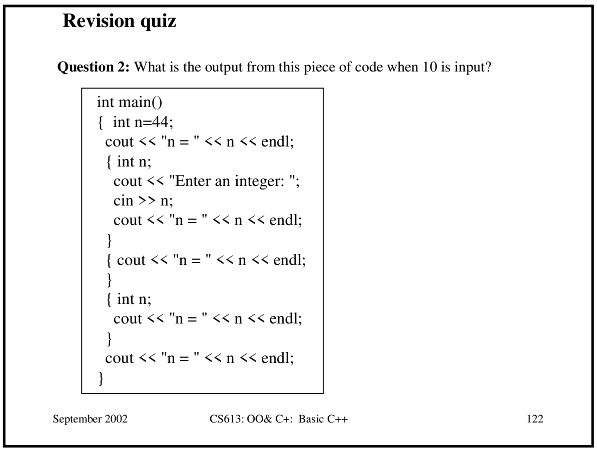


Revision quiz

Question 1: What is the output from this piece of code?

- A) When the input is 22 **22 = 22**
- B) When the input is 21 **22 = 22**

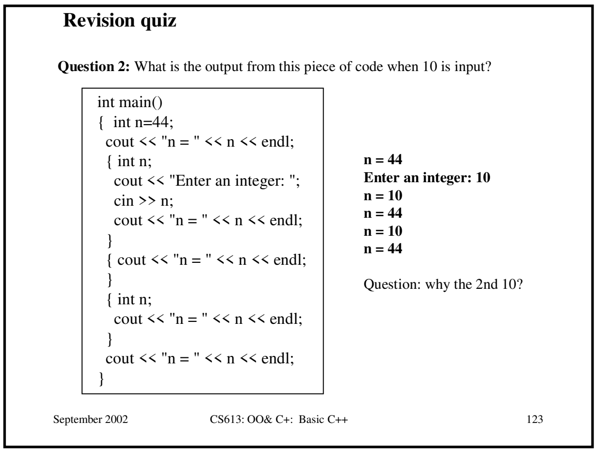
```
int main()
{int n;
  cout << "Enter an integer: ";
  cin >> n;
  if (n = 22) cout << n << " = 22" << endl;
  else cout << n << " != 22" << endl;
}
```



Revision quiz

Question 2: What is the output from this piece of code when 10 is input?

```
int main()
{  int n=44;
  cout << "n = " << n << endl;
  { int n;
    cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
  }
  { cout << "n = " << n << endl;
  }
  { int n;
    cout << "n = " << n << endl;
  }
  cout << "n = " << n << endl;
}
```



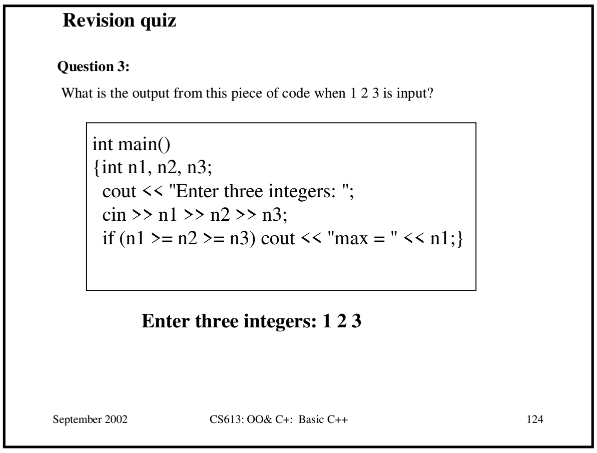
Revision quiz

Question 2: What is the output from this piece of code when 10 is input?

```
int main()
{  int n=44;
  cout << "n = " << n << endl;
  { int n;
    cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
  }
  { cout << "n = " << n << endl;
  }
  { int n;
    cout << "n = " << n << endl;
  }
  cout << "n = " << n << endl;
}
```

```
n = 44
Enter an integer: 10
n = 10
n = 44
n = 10
n = 44
```

Question: why the 2nd 10?



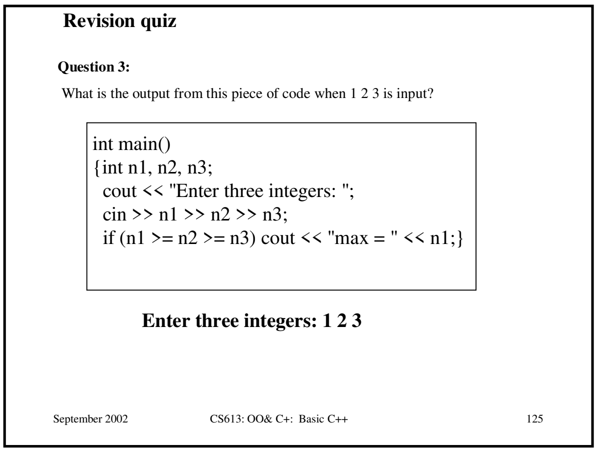
Revision quiz

Question 3:

What is the output from this piece of code when 1 2 3 is input?

```
int main()
{int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 >= n2 >= n3) cout << "max = " << n1;}
```

Enter three integers: 1 2 3



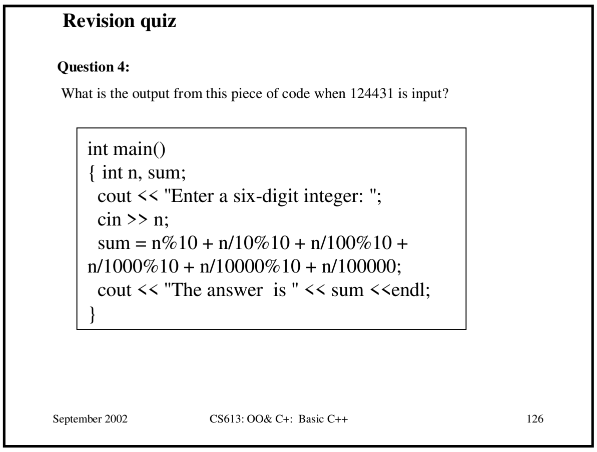
Revision quiz

Question 3:

What is the output from this piece of code when 1 2 3 is input?

```
int main()
{int n1, n2, n3;
  cout << "Enter three integers: ";
  cin >> n1 >> n2 >> n3;
  if (n1 >= n2 >= n3) cout << "max = " << n1;}
```

Enter three integers: 1 2 3

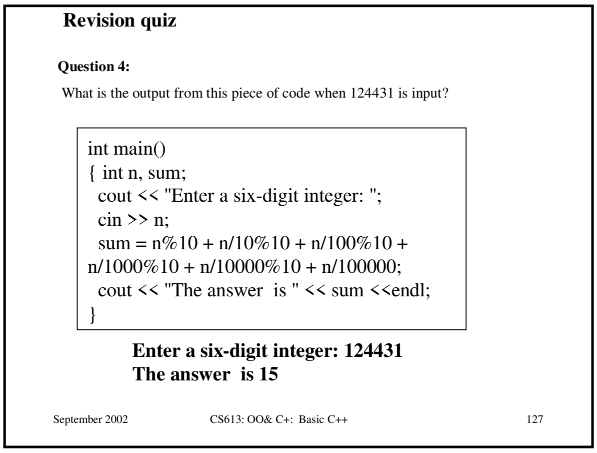


Revision quiz

Question 4:

What is the output from this piece of code when 124431 is input?

```
int main()
{ int n, sum;
  cout << "Enter a six-digit integer: ";
  cin >> n;
  sum = n%10 + n/10%10 + n/100%10 +
n/1000%10 + n/10000%10 + n/100000;
  cout << "The answer  is " << sum <<endl;
}
```



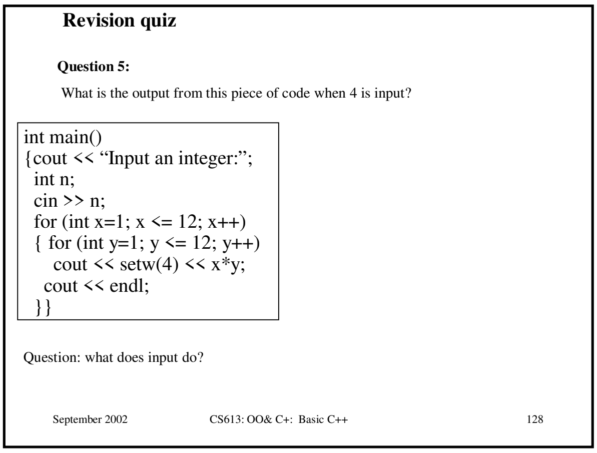
Revision quiz

Question 4:

What is the output from this piece of code when 124431 is input?

```
int main()
{ int n, sum;
  cout << "Enter a six-digit integer: ";
  cin >> n;
  sum = n%10 + n/10%10 + n/100%10 +
n/1000%10 + n/10000%10 + n/100000;
  cout << "The answer  is " << sum <<endl;
}
```

```
Enter a six-digit integer: 124431
The answer  is 15
```



Revision quiz

Question 5:

What is the output from this piece of code when 4 is input?

```
int main()
{cout << “Input an integer:”;
  int n;
  cin >> n;
  for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
     cout << setw(4) << x*y;
    cout << endl;
  }}
```

Question: what does input do?

Revision quiz

Question 5:

What is the output from this piece of code when 4 is input?

```
int main()
{cout << “Input an integer:”;
  int n;
  cin >> n;
  for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
      cout << setw(4) << x*y;
    cout << endl;
  }}
```

```
Input an integer:4
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
  12  24  36  48  60  72  84  96 108 120 132 144
```

Question: what does input do?

Revision quiz

Question 6:

What is the output from this piece of code when 70 is input?

```
int main()
{ float x;
  cout << "Enter a positive number: ";
  cin >> x;
  int n = 1;
  while (n*n <= x) ++n;
  cout << "The answer  is "
       << n-1 << endl;
}
```

Revision quiz

Question 6:

What is the output from this piece of code when 70 is input?

```
int main()
{ float x;
  cout << "Enter a positive number: ";
  cin >> x;
  int n = 1;
  while (n*n <= x) ++n;
  cout << "The answer  is "
       << n-1 << endl;
}
```

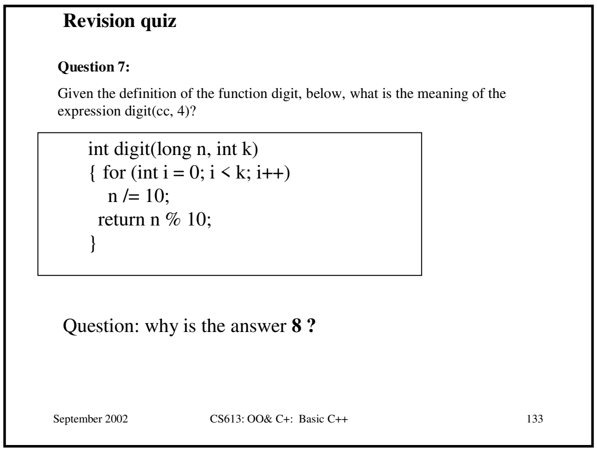
```
Enter a positive number:
70
The answer  is 8
```

Revision quiz

Question 7:

Given the definition of the function digit, below, what is the meaning of the expression digit(24681012, 4)?

```
int digit(long n, int k)
{ for (int i = 0; i < k; i++)
    n /= 10;
  return n % 10;
}
```



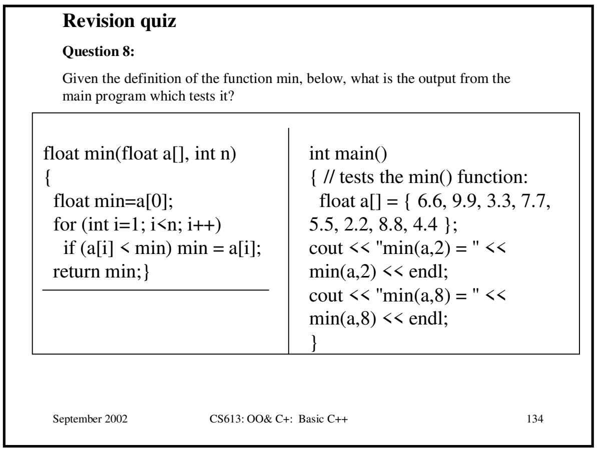
Revision quiz

Question 7:

Given the definition of the function digit, below, what is the meaning of the expression digit(cc, 4)?

```
int digit(long n, int k)
{ for (int i = 0; i < k; i++)
    n /= 10;
  return n % 10;
}
```

Question: why is the answer **8 ?**



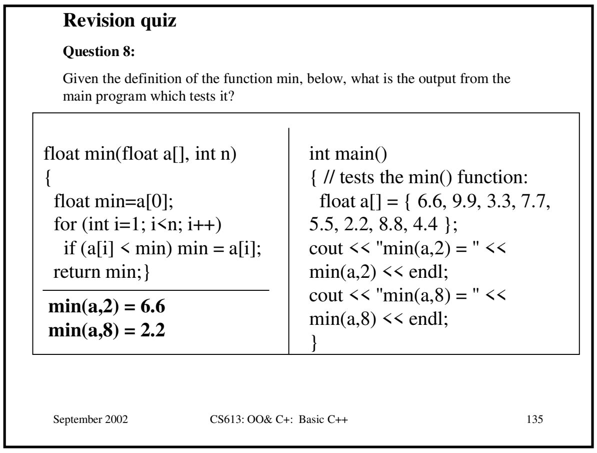
Revision quiz

Question 8:

Given the definition of the function min, below, what is the output from the main program which tests it?

```
float min(float a[], int n)
{
  float min=a[0];
  for (int i=1; i<n; i++)
    if (a[i] < min) min = a[i];
  return min;}
```

```
int main()
{ // tests the min() function:
  float a[] = { 6.6, 9.9, 3.3, 7.7,
5.5, 2.2, 8.8, 4.4 };
cout << "min(a,2) = " <<
min(a,2) << endl;
cout << "min(a,8) = " <<
min(a,8) << endl;
}
```



Revision quiz

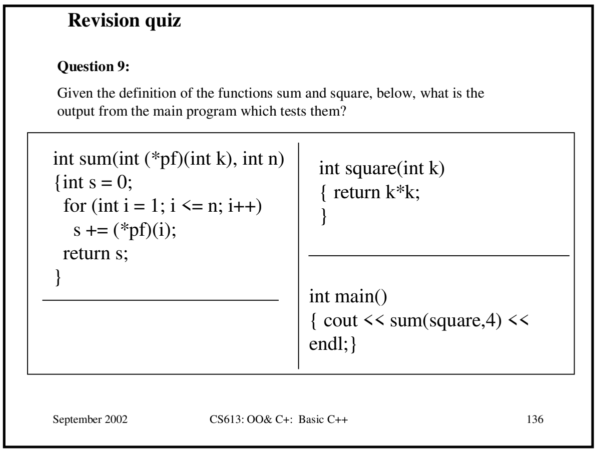
Question 8:

Given the definition of the function min, below, what is the output from the main program which tests it?

```
float min(float a[], int n)
{
  float min=a[0];
  for (int i=1; i<n; i++)
    if (a[i] < min) min = a[i];
  return min;}
```

min(a,2) = 6.6
min(a,8) = 2.2

```
int main()
{ // tests the min() function:
  float a[] = { 6.6, 9.9, 3.3, 7.7,
5.5, 2.2, 8.8, 4.4 };
cout << "min(a,2) = " <<
min(a,2) << endl;
cout << "min(a,8) = " <<
min(a,8) << endl;
}
```



Revision quiz

Question 9:

Given the definition of the functions sum and square, below, what is the output from the main program which tests them?

```
int sum(int (*pf)(int k), int n)
{int s = 0;
  for (int i = 1; i <= n; i++)
    s += (*pf)(i);
  return s;
}
```

```
int square(int k)
{ return k*k;
}
```

```
int main()
{ cout << sum(square,4) <<
endl;}
```

Revision quiz

Question 9:

Given the definition of the functions sum and square, below, what is the output from the main program which tests them?

```
int sum(int (*pf)(int k), int n)
{int s = 0;
  for (int i = 1; i <= n; i++)
    s += (*pf)(i);
  return s;
}
```

30

```
int square(int k)
{ return k*k;
}
```

```
int main()
{ cout << sum(square,4) <<
endl;}
```

Revision quiz

Question 10:

Given the definition of the cap function, below, what is the output from the main program which tests it, when the input is “J paul Gibson”?

```
void cap(char* s)
{ if (s == NULL) return;
  for (char* p = s; *p; p++)
    if (*p>='a' && *p<='z')
       *p = (char)(*p-'a'+'A');
}
```

J paul Gibson
Answer is J PAUL GIBSON

```
int main()
{ char *line = new char[81];
  cout << "Enter line of text
below:" << endl;
  cin.getline(line,80);
  cap(line);
  cout << ”Answer is “<< line
<< endl;
}
```

Revision quiz

Question 10:

Given the definition of the cap function, below, what is the output from the main program which tests it, when the input is “J paul Gibson”?

```
void cap(char* s)
{ if (s == NULL) return;
  for (char* p = s; *p; p++)
    if (*p>='a' && *p<='z')
       *p = (char)(*p-'a'+'A');
}
```

```
int main()
{ char *line = new char[81];
  cout << "Enter line of text
below:" << endl;
  cin.getline(line,80);
  cap(line);
  cout << ”Answer is “<< line
<< endl;
}
```

Revision quiz

Question 10:

Given the definition of the cap function, below, what is the output from the main program which tests it, when the input is “J paul Gibson”?

```
void cap(char* s)
{ if (s == NULL) return;
  for (char* p = s; *p; p++)
    if (*p>='a' && *p<='z')
       *p = (char)(*p-'a'+'A');
}
```

jPaulGibson
Answer is JPAULGIBSON

```
int main()
{ char *line = new char[81];
  cout << "Enter line of text
below:" << endl;
  cin.getline(line,80);
  cap(line);
  cout << ”Answer is “<< line
<< endl;
}
```

Noughts and Crosses - putting it ‘all together

The (chosen/given) data structure :-

```
        const boardsize =3;
        typedef char XOBoard [boardsize][boardsize];
```

Example of use:

```
        int main()
        {
        XOBoard board;
        initialise (board);
        print(board);
        playX(board, 1,1);
        print(board);
        playO(board, 0,0);
        print(board);
        }
```

Questions:

good/bad structure?

functions required?

Noughts and Crosses

The (chosen/given) data structure :-

```
        const int boardsize =3;
        typedef char XOBoard [boardsize][boardsize];
```

Example of use:

```
        void initialise (XOBoard);
        void print (XOBoard);
        void playX(XOBoard, int, int);
        void playO(XOBoard, int, int);

        int main()
        { \\ As before
        }
```

GOOD PROGRAMMING:

•put function type declaration at beginning of file before the main program

•put function definitions after the main program

Noughts and Crosses

Design and test one function at a time (if possible)

initialise -

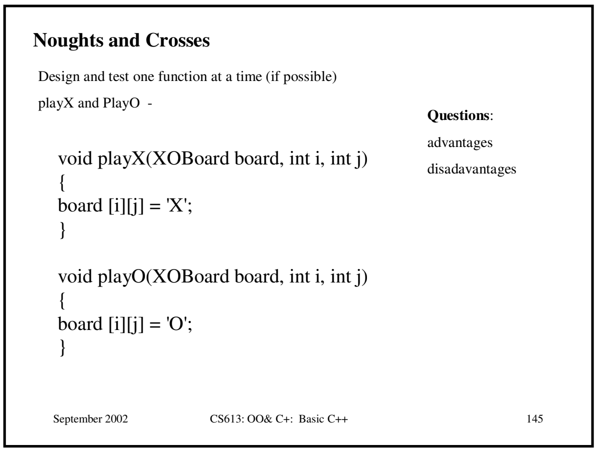
```
void initialise(XOBoard board)
{
for (int i = 0; i<3; i++)
  for (int j = 0; j<3; j++)
     board [i][j] = ' ';
}
```

Noughts and Crosses

Design and test one function at a time (if possible)

print -

```
void print (XOBoard board)
{
cout << endl;
for (int i = 0; i<3; i++)
  { for (int j = 0; j<3; j++)
     { cout << board [i][j];
       if (j<2) cout << "|";
     }
    cout << endl;
    if (i<2) cout << "-----" << endl;
  }
}
```



Noughts and Crosses

Design and test one function at a time (if possible)

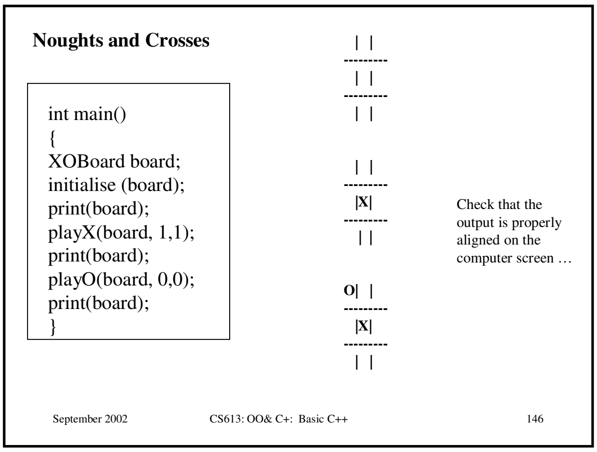
playX and PlayO -

```
void playX(XOBoard board, int i, int j)
{
board [i][j] = 'X';
}

void playO(XOBoard board, int i, int j)
{
board [i][j] = 'O';
}
```

Questions:

advantages

disadavantages

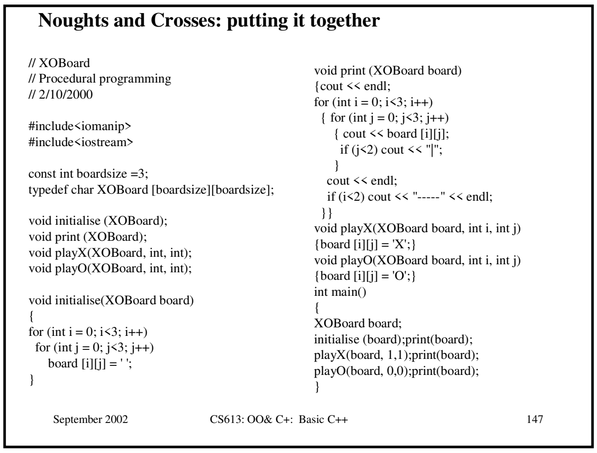


Noughts and Crosses

```
int main()
{
XOBoard board;
initialise (board);
print(board);
playX(board, 1,1);
print(board);
playO(board, 0,0);
print(board);
}
```

```
 | |
---------
 | |
---------
 | |

 | |
---------
 |X|
---------
 | |

O| |
---------
 |X|
---------
 | |
```

Check that the output is properly aligned on the computer screen …



Noughts and Crosses: putting it together

```
// XOBoard
// Procedural programming
// 2/10/2000

#include<iomanip>
#include<iostream>

const int boardsize =3;
typedef char XOBoard [boardsize][boardsize];

void initialise (XOBoard);
void print (XOBoard);
void playX(XOBoard, int, int);
void playO(XOBoard, int, int);

void initialise(XOBoard board)
{
for (int i = 0; i<3; i++)
  for (int j = 0; j<3; j++)
     board [i][j] = ' ';
}
```

```
void print (XOBoard board)
{cout << endl;
for (int i = 0; i<3; i++)
  { for (int j = 0; j<3; j++)
     { cout << board [i][j];
       if (j<2) cout << "|";
     }
    cout << endl;
    if (i<2) cout << "-----" << endl;
  }}
void playX(XOBoard board, int i, int j)
{board [i][j] = 'X';}
void playO(XOBoard board, int i, int j)
{board [i][j] = 'O';}
int main()
{
XOBoard board;
initialise (board);print(board);
playX(board, 1,1);print(board);
playO(board, 0,0);print(board);
}
```