I don’t know what the programming language of the year 2000 will look like, but I know it will be called FORTRAN.
iostream and namespace

- Include `iostream` for input/output.
- Then, add `using namespace std;`

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

int main(...) {
    // other C/C++ statements
}
```
Input with `cin` and `>>`

- Use `cin` and `>>` to read from `stdin`.
- For example, `cin >> n` reads in a data item from `stdin` to variable `n`.
- One more example: `cin >> a >> b` reads in two data items from `stdin` to variables `a` and `b` in this order.
- Thus, `cin` is easier to use than `scanf`.
Output with cout and << : 1/2

- Use `cout` and `<<` to write to `stdout`.
- For example, `cout << n` writes the content of variable `n` to `stdout`.
- One more example: `cout << a << b` writes the values of variables `a` and `b` to `stdout` in this order.
- Thus, `cout` is easier to use than `printf`.
- Formatted output with `cout` is very tedious.
Output with `cout` and `<<` : 2/2

- The `\n` is `endl`: `cout << a << endl` prints the value of `a` and follows by a newline.
- You may want to add spaces to separate two printed values.
- `cout << a << ' ' << b << endl` is better than `cout << a << b << endl`.


#include <iostream>

using namespace std;

int main(void)
{
    cout << "Hello, world." << endl;
    return 0;
}
#include <iostream>  
using namespace std;

int main(void)
{
    int i, n, factorial;

    cout << "A positive integer --> ";
    cin >> n;
    factorial = 1;
    for (i = 1; i <= n; i++)
        factorial *= i;
    cout << "Factorial of " << n << " = "
        << factorial << endl;
    return 0;
}
What Is a **class**?

- A **class** is a type similar to a **struct**; but, a **class** type normally has member functions and member variables.

```cpp
class Sum_and_Product
{
    public:
        int a, b;
        void Sum(), Product();
        void Reset(int, int), Display();
    private:
        int MySum, MyProduct;
};
```
Constructors : 1/2

- Constructors are member functions and are commonly used to initialize member variables in a class.
- A constructor is called when its class is created.
- A constructor has the same name as the class.
- A constructor definition cannot return a value, and no type, not even void, can be given at the beginning of the function or in the function header.
Constructors : 2/2

- Constructors are commonly used to initialize member variables in a class.

```cpp
class MyClass
{
    public:
        MyClass(int n);  // constructor
    // ...
};
MyClass::MyClass(int Input) // function
{
    // ...
}
```
Member Functions

- Member functions are just functions.

```cpp
class MyClass
{
    public:
        MyClass(int n);  // constructor
        void Display(...); // member function
        // ...
};

MyClass::Display(...) // function
{
    // ......
}
```
Example: 1/5

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

class MyAccount
{
    public:
        MyAccount(int Initial_Amount); // constructor
        int Deposit(int);             // member funct
        int Withdraw(int);            // member funct
        void Display(void);           // member funct

    private:
        int Balance;                  // private variable
};
```

Example: 2/5

MyAccount::MyAccount(int initial)
{
    Balance = initial;  // constructor initialization
}

int MyAccount::Deposit(int Amount)
{
    cout << "Deposit Request = " << Amount << endl;
    cout << "Previous Balance = " << Balance << endl;
    Balance += Amount;
    cout << "New Balance = " << Balance << endl;
    return Balance;
}
Example: 3/5

```cpp
int MyAccount::Withdraw(int Amount) {  
    cout << "Withdraw Request = " << Amount << endl;  
    cout << "Previous Balance = " << Balance << endl;  
    Balance -= Amount;  
    cout << "New Balance     = " << Balance << endl  
        << endl;  
    return Balance;  
}

void MyAccount::Display(void) {  
    cout << "Current Balance  = " << Balance << endl  
        << endl;  
}
```
int main(void)
{
    MyAccount NewAccount(0); // initial new account

    NewAccount.Display();       // display balance
    NewAccount.Deposit(20);     // deposit 20 (Bal=20)
    NewAccount.Deposit(35);     // deposit 35 (Bal=55)
    NewAccount.Withdraw(40);    // withdraw 40 (Bal=15)
    NewAccount.Display();       // current balance
    return 0;
}
```c
int main(void)
{
    MyAccount *NewAccount;        // use pointer
    NewAccount = new MyAccount(0); // create account
    NewAccount->Display();         // now use ->
    NewAccount->Deposit(20);
    NewAccount->Deposit(35);
    NewAccount->Withdraw(40);
    NewAccount->Display();
    return 0;
}
```

This version uses a pointer.
The `new` operator creates an object and returns a pointer to it.
It is similar to `malloc()` in C. Use `delete` to deallocate.
There is a faster way, actually maybe a preferable way, to initialize member variables.

```cpp
class Numbers
{
    public:
        int Lower, Upper;
        Numbers(int a, int b);     // constructor
        // ...
};

Numbers::Numbers(int a, int b)
    : Lower(a), Upper(b)     // init. section
{  // function body is empty
}
```
Derived Classes: 1/6

- Deriving a class from an existing one is called *inheritance* in C++.
- The newly created class is a *derived* class and the class from which the derived class is created is a *base* class.
- The constructor (and destructor) of a base class is not inherited.
A derived class is just a class with the following syntax:

```cpp
class derived-class-name : public base-class-name {
    public:
        // public member declarations
        derived-class-constructor();
    private:
        // private member declarations
};
```
class Base
{
    public:
        int a;
        Base(int x=10):a(x) // use x to init a
            { cout << "Base has " << a << endl; } 
    
};

class Derived: public Base 
{
    public:
        int x;
        Derived(int m=20):x(m) // use m to init x
            { cout << "Derived has " << x << endl; } 

};
int main(void) {
    Base X, *XX;
    Derived Y, *YY;
    
    cout << "Base's value = " << X.a << endl;
    cout << "Derived's value = " << Y.x << endl;
    cout << endl;
    XX = new Base(123);
    YY = new Derived(789);
    cout << "Base's value = " << XX->a << endl;
    cout << "Derived's value = " << YY->x << endl;
    return 0;
}

X.a = 10, Y.x = 20
XX->a = 123, YY->x = 789
class Base
{
    public:
        int  a;
        char name[100];
    Base(int);
};
Base::Base(int x = 10) : a(x)
{
    char  buffer[10];
    strcpy(name, "Class");    // requires string.h
    sprintf(buffer, "%d", a); // requires stdio.h
    strcat(name, buffer);     // requires string.h
    cout << "Base has " << a << ' ' << name << endl;
}
Derived Classes: 6/6

class Derived: public Base
{
    public:
        Derived(int m=20): Base(m) {  }
};

int main(void)
{
    Base     X(23);
    Derived  Y(789);

    cout << "Base's name    = " << X.name << endl;
    cout << "Derived's name = " << Y.name << endl;

    return 0;
}
Normally, the specification part and the implementation part of a class are saved in `.h` and `.cpp` files, respectively.

```cpp
class MyAccount
{
    public:
        MyAccount(int Initial_Amount);
        int Deposit(int);
        int Withdraw(int);
        void Display(void);

    private:
        int Balance;
};
```
```cpp
#include <iostream>
#include "MyAccount.h"

using namespace std;

MyAccount::MyAccount(int initial)
    : Balance(initial)
{ /* function body is empty */ }

int MyAccount::Deposit(int Amount)
{
    cout << "Deposit Request = " << Amount << endl;
    cout << "Previous Balance = " << Balance << endl;
    Balance += Amount;
    cout << "New Balance = " << Balance << endl;
    return Balance;
}
// other member functions
```
#include <iostream>
#include  "MyAccount.h"

using namespace std;

int main(void)
{
    MyAccount  *NewAccount;

    NewAccount = new MyAccount(0);
    NewAccount->Display();
    NewAccount->Deposit(20);
    NewAccount->Deposit(35);
    NewAccount->Withdraw(40);
    NewAccount->Display();
    return 0;
}
Now we have the specification file `MyAccount.h`, the implementation file `MyAccount.cpp`, and the main program file `account-3.cpp`.

Compile the whole thing this way:

```
g++ MyAccount.cpp account-3.cpp -o account-3
```

Or, we may compile `MyAccount.cpp` to `MyAccount.o` and use it later:

```
g++ MyAccount.cpp -c

```
```
g++ account-3.cpp MyAccount.o -o account-3
```
ThreadMentor Architecture

- **ThreadMentor** consists of a class library and a visualization system.
- The class library provides all mechanisms for thread management and synchronization primitives.
- The visualization system helps visualize the dynamic behavior of multithreaded programs.
ThreadMentor Architecture

- C++ User Program
- Synchronization
- Thread Kernel
  - Win32
  - Solaris
  - Pthread
  - mtuThread

Visualization
Basic Thread Management

- **Thread creation**: creates a new thread
- **Thread termination**: terminates a thread
- **Thread join**: waits for the completion of another thread
- **Thread yield**: yields the execution control to another thread
- **Suspend/Resume**: suspends or resumes the execution of a thread.
How to Define a Thread?

- A thread should be declared as a derived class of `Thread`.
- All executable code must be in function `ThreadFunc()`.
- A thread may be assigned a name with a constructor.
- Method `Delay()` may be used to delay the thread execution for a random time.

```cpp
#include "ThreadClass.h"
class test : public Thread
{
    public:
        test(int i){n=i;};
    private:
        int n;
        void ThreadFunc(int);
};
void test::ThreadFunc(int n)
{
    Thread::ThreadFunc();
    for (int i=0; i<10; i++)
    {
        cout << n << i << endl;
        // other stuffs
    }
}
```

may not be thread safe!
Create and Run a Thread

- Declare a thread just like declaring an `int` variable.
- Then, use method `Begin()` to run a thread.

```c
int main(void)
{
    test* Run[3];
    int i;
    for (i=0; i<3; i++) {
        Run[i] = new test(i);
        Run[i]->Begin();
    }
    // other stuffs
}
```
A Few Important Notes

- Before calling method `Begin()`, the created thread *does not* run.
- Function `ThreadFunc()` *never* returns. When it reaches the end or executes a return, it *disappears*!
- Do not use `exit()`, as it terminates the whole system. See next slide.
Terminating a Thread

- Use method `Exit()` of the thread class `Thread`.
- Do not use system call `exit()` as it terminates the whole program.

```cpp
void test::ThreadFunc(int n) {
    Thread::ThreadFunc();
    for (int i=0; i<10; i++)
        cout << n << i << end;
    Exit(); // terminates
}
```
Thread Join

- Sometimes, a thread must wait until the completion of another thread so that the results computed by the latter can be used.

- The parent must wait until all of its child threads complete. Otherwise, when the parent exits, all of its child threads exit.
The Join() Method

- Use the `Join()` method of a thread to join with that thread.

- Suppose thread A must wait for thread B’s completion. Then, do the following in thread A:
  
  ```
  B->Join()
  ```

  or

  ```
  B.Join()
  ```
Thread Join Semantics

Suppose thread A wants to join with thread B, we have two cases:

1. If A reaches the \texttt{Join()} call before B exits, A waits until B completes.
2. If B exits before A can reach the \texttt{Join()} call, then A continues as if there is no \texttt{Join()}. 
A Simple Example

```cpp
#include "ThreadClass.h"

class test : public Thread
{
    public:
        test(int i){n = i;};
    private:
        int n;
        void ThreadFunc();
};

test::ThreadFunc(int n)
{
    Thread::ThreadFunc();
    for (int i=0; i<10; i++)
        cout << n << i << endl;
    Exit();
}

test* Run[3];

int main(void)
{
    for (int i=0;i<3;i++)
    {    
        Run[i] = new test(i);
        Run[i]->Begin();
    }
    for (i = 0; i<3; i++)
        Run[i]->Join();
    Exit();
    May not be thread safe.
    Why?
}
```
Threaded Quicksort: 1/3

- In each recursion step, the quicksort cuts the given array segment $a[L:U]$ into two with a pivot element $a[M]$ such that all elements in $a[L:M-1]$ are less than $a[M]$ and all elements in $a[M+1:U]$ are greater than $a[M]$. Then, $a[L:M-1]$ and $a[M+1:U]$ are sorted independently and recursively.

- Since $a[L:M-1]$ and $a[M+1:U]$ are sorted independently, we may use a thread for each segment!
**Threaded Quicksort: 2/3**

- A thread receives the array segment $a[L:U]$ and partitions it into $a[L:M-1]$ and $a[M+1:U]$.
- Then, creates a thread to sort $a[L:M-1]$ and a second thread to sort $a[M+1:U]$.
**Threaded Quicksort: 3/3**

Thus, our strategy looks like the following:

2. It finds the pivot element $a[M]$.
3. Creates a child thread and provides it with $a[L:M-1]$.
5. Issues two thread `Join()`s waiting for both child threads.
Class **Quicksort**: Definition

```cpp
class Quicksort : public Thread
{
    public:
        Quicksort(int L, int U, int a[]);
    private:
        int  low;
        int  up;
        int  *a;
        void  ThreadFunc();
};
```
Class **Quicksort**: Implementation

```cpp
Quicksort::Quicksort(int L, int U, int A[]) : low(L), up(U), a(A) {
    ThreadName = // set a thread name;
}

Void Quicksort::ThreadFunc() {
{
    Thread::ThreadFunc(); // required
    Quicksort  *Left, *Right;
    int M;
    M = // compute the pivot element;
    Left = new Quicksort(low, M-1, a);  Left->Begin();
    Right = new Quicksort(M+1, up, a); Right->Begin();
    Left->Join(); Right->Join();
    Exit();
}
```
The main program is easy:

```c
int main(void)
{
    Quicksort *thread;
    int a[MAXSIZE], L, U, n;
    // read in array a[] and # of elements n
    L = 0; U = n-1;
    thread = new Quicksort(L, U, a);
    thread->Begin();
    thread->Join();
    Exit();
}
```
What If We Have the Following?

```c++
Quicksort::Quicksort(int L, int U, int A[])
: low(L), up(U), a(A)
{
    ThreadName = // set a thread name;
}

Void Quicksort::ThreadFunc()
{
    Thread::ThreadFunc();
    Quicksort *Left, *Right;
    int M;
    M = // compute the pivot element;
    Left = new Quicksort(low, M-1, a);
    Left->Begin(); Left->Join();
    Right = new Quicksort(M+1, up, a);
    Right->Begin(); Right->Join();
    Exit();
}
```

Join() are moved to right after Begin(). Is this a correct program? Does it fulfill the maximum concurrency requirement?
Compilation with ThreadMentor

- *ThreadMentor* adds all visualization features in its class library so that you don’t have to do anything in your program to use visualization.

- But, you need to recompile your program properly so that a correct library will be used.

- There are two versions of *ThreadMentor* library: Visual and non-Visual.
Makefile for ThreadMentor: 1/4

CC       = c++
CFLAGS   = -g -O2
DFLAGS   = -DPACKAGE="threadsystem" ......
IFLAGS   = -I/local/eit-linux/apps/ThreadMentor/include
TMLIB    = /local/eit-linux/apps/ThreadMentor/Visual/......
TMLIB_NV = /local/eit-linux/apps/ThreadMentor/NoVisual/......

OBJ_FILE = quicksort.o quicksort-main.o
EXE_FILE = quicksort

Define some names. Don’t touch this portion.

This is the executable file
List the .o files here
${EXE_FILE} : ${OBJ_FILE}

${CC} ${FLAGS} -o ${EXE_FILE} ${OBJ_FILE} ${TMLIB} -lpthread

quicksort.o : quicksort.cpp
   ${CC} ${DFLAGS} ${IFLAGS} ${CFLAGS} -c quicksort.cpp

quicksort-main.o : quicksort-main.cpp
   ${CC} ${DFLAGS} ${IFLAGS} ${CFLAGS} -c quicksort-main.cpp

noVisual : ${OBJ_FILE}
   ${CC} ${FLAGS} -o ${EXE_FILE} ${OBJ_FILE} ${TMLIB_NV} -lpthread

clean:
   rm -f ${OBJ_FILE} ${EXE_FILE}
Makefile for ThreadMentor: 3/4

- By default, the above `Makefile` generates executable with visual. The following generates executable `quicksort`:
  ```
  make
  ```
- If you do not want visualization, use the following:
  ```
  make noVisual
  ```
- To clean up the `.o` and executable files, use
  ```
  make clean
  ```
Add the following line to your `.cshrc`, which is in your home directory. Then, logout and login again to make it effective:

```
set path=($path /local/eit-linux/apps/ThreadMentor/bin)
```

More *ThreadMentor* examples are available at the *ThreadMentor* tutorial site:

The End