Enhanced C, and a language for Object-Oriented Programming
Traditional vs. Object-Oriented Programming Paradigm

- **Traditional Paradigm**
  - Algorithm identification
  - Functional decomposition:
    - Control Flow construction
    - Data Structure required for functions

- **Object-Oriented Paradigm**
  - Object identification
    - ADTs that represent the problem domain
  - Object Characteristic definitions
    - Internal attributes of object
  - Object Method (operation) definitions
    - Abstract operations (i.e., interface) for the ADTs
  - Solution is a sequence of calls to methods
Why OOP?

- The objects are the most stable factor in the problem
  - Algorithms may be improved
  - Implementations may be modified
  - Interface may undergo major changes

- If the data structure for the objects is defined according to algorithm / implementation / interface, it is very likely to backfire when one of them is changed

- Objects are the appropriate level at which decisions about encapsulation (information hiding) are made
Interface vs. Implementation

What to do
Message
Method
How to do

Introduction to Systems Programming
In Object Oriented Programming, the message “rotate (around the center) by \( n \) degrees clockwise” will be implemented differently by different objects:

**Shape**: Do nothing if \( n = 360k \), otherwise: rotate . . .

**Circle**: Do nothing.

**Square**: Do nothing if \( n = 90k \), otherwise: rotate . . .

The crucial part is that all may appear to the client as “shape”, but the correct operation will be invoked.
OOP Will

- Encourage use of ADTs and their extensions (classes, modules, packages, ...)
- Make design decisions easier
- Encourage code reuse (within same/other project(s))

The above will result in
- Making code predictable to use
- Ease program implementation
- Ease understanding of program structure
- Ease program maintenance
  - Corrections
  - Modifications
  - Enhancements
- Help in documentation
OOP Will NOT

- Design your program for you
- Prevent you from designing a bad program
- Provide algorithms
- Perform data management
File complex.h:
```c
typedef struct {
    double re, im;
} Complex;
Complex ComplexCreate (double r = 0., double i = 0.);
```

File complex.c:
```c
Complex ComplexCreate (double r, double i) {
    Complex c;  // re and im are undefined.
    c.re = r;
    c.im = i;
    return c;   // return a copy of c.
}
```

Function parameters may be given default values at declaration

The default is used if the parameter is not supplied
Call by Reference

1. Let the compiler do the work:

```c
void swap (int& a, int& b)
{ int t = a; a = b; b = t; }

swap(x,y); // OK
swap(5,y); // Error
```

2. Pass by reference large data:

```c
int sum (const large_data&);
```

3. Left/right-hand-side expressions:

```c
double& Index
  (double a[], int indx)
{ return a[indx]; }
```

```c
y = index (vec,j); // same as y = vec[j];
index(vec,i) = 5.2; // same as vec[i]=5.2;
```

- T& is a memory location of type T.
- Can serve as both lhs or rhs in an assignment, taking the address or value, respectively
- An expression of type T& can serve anywhere an expression of type T can, but not vice-versa
- Used when a value of an argument is to be modified
- Also used for efficiency, as only address of location (a pointer) is passed, not value itself
- In the latter case, the keyword `const` is used to prevent the object from being modified
Function Prototyping & Overloading

```c
void swap (int&, int&);
void swap (double&, double&);

int a, b;
double c, d;

swap (a, b);  // calls first swap
swap (c, d);  // calls second swap
swap (a, c);  // syntax error (no conversion for references)
```

- A function is identified by its entire prototype (signature = list of parameters), not just by its name
- The function (if there is any) whose entire prototype matches best the calling statement will be invoked
Advantages:
- Global constants
- Common functionality without function call overhead
- Common functionality on variety of types

Disadvantages:
- Do not obey scope rules
- Type-unsafe constructs

Note: `enum` types can be defined for int constants only; for other types we need `const` variables for which memory is allocated, unlike macros.

C++ alternatives:
- `enum` and `const` types: For replacing MACRO constants
- `inline` functions: For eliminating function-call overhead
- `template` functions: For type-safe common functionality on different types
Inline Functions

- The compiler will generate in-line code, hence there will be no function-call overhead:
  
  ```
  inline int max (int a, int b) {
    return a < b ? b : a;
  }
  ```

- The function body **must** appear in each file where it is used. Hence, it is usually given in a header-file and `#included` where needed.

- Obviously, an in-line function cannot be recursive.

- In fact, the compiler is not obliged to substitute the function call by its body (say, if the function is too looooooong), but it usually will.

- Caveat: cannot be used for evaluating `max()` of two doubles.

- However, the `max` macro is not fool-proof either (e.g., when used on two `char*` objects, or for types without the operator “<”).

- Full solution will be template functions (later in this course).
Suppose we have two versions of the function \texttt{sqrt()}, one is implemented for speed and the other for accuracy. Furthermore, we have a huge software which uses this function, and we want to control easily which of the two versions of the function the entire software uses. In C, we probably have them in two libraries, and we will link the software with the appropriate library. But then we may get the wrong version of \texttt{pow()}. Moreover, what can we do if we want to use both versions in the same software?

```cpp
namespace std { 
    double sqrt(double);
    double pow (double, double);
}

namespace alt { 
    double sqrt(double);
    double pow (double, double);
}

double pyth(double x, double y) 
{
    return std::sqrt(x*x + y*y);
}

double PDE_solver(PDE e) {
    ...
    var1 = alt::sqrt(var2);
    ...
}
```
Namespaces can be nested
Namespaces can be extended.
Global access to a specific identifier in a namespace is achieved by a using declaration.
Global access to all identifiers in a namespace is achieved by a using directive.
Caution: using directives should be used judiciously and never in headers!
Identifiers within an unnamed namespace are available in the same translation unit only.

Ideally, namespaces should
* Express a logical coherent set of features
* Prevent user access to unrelated features
* Impose minimal notational burden on users

(B.S. C++PL 3rd ed.)
C++ provides tools for supporting a safe handling of object types:

- Operations on objects of different types (e.g., assignment, binary ops) are prohibited unless their semantics is clear.
- Conversion of objects from one type to another are supported by constructors and templates.
- “Constness” of objects is enforced when needed.
- See further slides...
Unlike C, C++ prevents in most cases unsafe type-conversions.

The usage of casting is **discouraged**, and is considered a safety problem as well as a bad taste.

The same is true for type `void*`.

C++ provides special purpose casting operators, that are much safer.

```c
int i;
double x;

x = i; // o.k. (usually)
i = x; // warning (usually)
```

```c
int *pi;
void *ptr;

ptr = pi; // o.k.
pi = ptr; // syntax error !!
```
A special treatment is given to `const` objects:

- They cannot be modified; hence, they must be initialized.
- They cannot be sent to a function that *may* modify them; hence, a function must declare whether it modifies an argument.
- An argument that is passed to a function by value is never modified by a function. That is, the function can modify only its *copy* but not the original object.

```c
char *str = "abc"; // **Bad** (str[0] = ‘x’; is possible)
char const *cst = "abc"; // o.k. – the only safe way
void func(char *); // a function declaration
func(cst); // syntax error at a function call

const char * const cst = "xyz"; // two different “const”s!
```

Enumerated constants and `const` objects may replace MACROs that `#define` constants.

*A Warning (deprecated)*

[Introduction to Systems Programming]
### Strong Typing: `enum` Issues

```c
enum A { a, b, c};

int a = 3;    // Error: ‘a’ is redeclared (same as C)

int main()
{
    enum X { x, y, z};
    double a = 3.14;   // OK, inner scope (in both C and C++)
    double x = 2.71828; // Error: ‘x’ is redeclared (same as C)

    X x1 = 1;         // Error: invalid conversion from ‘int’ (OK in C!)
    A c = z;          // Error: invalid conversion from ‘X’ (OK in C!)
    int n = y;        // OK: enum2int conversion (in both C and C++)

    return 0;
}
```

- `X` and `A` are type-names.
Strong Typing: Scoped `enum`

```c
enum class A {a, b, c};  // The {} is a ‘scope’

int a = 3;  // OK

int main()
{
    enum class X {x, y, z};
    double a = 3.14;  // OK, inner scope (in both C and C++)
    double x = 2.71828;  // OK

    X x1 = 1;  // Error: invalid conversion from ‘int’ to ‘X’
    A c = X::z;  // Error: invalid conversion from ‘X’ to ‘A’
    int n = A::a;  // Error: invalid conversion from ‘A’ to ‘int’
    return 0;
}
```
Strong Typing: auto

Let the compiler find the exact type of an object. The type of an auto object is the type of the initializer, but in some circumstances it is not what you want (or wrong).

```c
Set my_set = SetCreate(...);
#define EXTRACT_LMNT(e) ...
// Like TO_INT(e)
SET_FOREACH(e, my_set) {
    auto lmnt = EXTRACT_LMNT(e);
}
auto n; // Error: missing initialization
auto sz = my_set->size(); // Is it int / unsigned / long unsigned ?
```

A C style example

Finds the exact type: Less typedef, const awareness, and much more

C++ version of SetSize(my_set)

```c
auto root = (float)(sqrt(5.0)); // casting
```