Classes

- Information hiding
- Guaranteed initialization and finalization
- Overloaded operators
- Dynamic typing
- User-controlled memory management

Other advantages:

- Overloaded function names
- Overloaded operators
- User-controlled memory management
- Dynamic typing
- Guaranteed initialization and finalization

References for C++

- Bjarne Stroustrup
  - *The C++ Programming Language (3rd ed.)*
  - Addison-Wesley, 1997
- Scott Meyers
  - *Effective C++ (3rd ed.)*
  - Addison-Wesley, 1998
- S. McConnell
  - *C++ Primer (3rd ed.)*
  - S. Lippman, 1996
- Hiroshi Hayashi
  - *Object-Oriented Programming in C++*
  - Prentice Hall, 1997
- David B. Wortman
  - *C++ Primer (3rd ed.)*

References for Reading Assignment

- Chapter 6, S. McConnell
- Chapter 19, K. N. King
- Chapter 7, D. B. Wortman
- Slides 278, 289

Supplementary Reading

- Chapter 9, D. B. Wortman
- Chapter 6, S. McConnell

Reading Assignment

- CSC 181F Lecture Notes
Default Function Arguments

You cannot skip over arguments.

The arguments are assigned to the corresponding parameters from left to right.

A parameter without defaults cannot occur after a parameter with defaults.

The defaults must be added from right to left.

```c
f(14);  // (14) 14.83 E0
f(14, 14.83);  // (14) (14) 14.83 E0
f(14, 14.83, 'n', NULL, 'msg', 12.6);  // (14) (14) 14.83 'n' 'E0', 'msg' = 12.6
```

Example:

```c
void F(int val, float S=12.6, char T=' ', char* msg="Error");

f(14, 48.3, 't', "OK");
f(14, 48.3, 't');
f(14, 48.3);
f(14);
```

The defaults must be added from right to left.

A parameter without defaults cannot occur after a parameter with defaults.

The arguments are assigned to the corresponding parameters from left to right; you cannot skip arguments for function arguments in the prototype.

Default values may be supplied for function arguments in the prototype.

### Function Parameters

#### Function with no argument in a function prototype

```c
int F();
```

is equivalent to

```c
int F(void);
```

#### Inline Functions

**inline** is a request that a function be expanded inline.

- Place the keyword `inline` before the function definition.
- Place the function definition above all the functions that call it.
- Place the keyword `inline` before the function definition in `#inline` functions.
- Specify no parameters.

#### Functions with no argument in a function prototype are interpreted as

```c
void inline float cube(float s)
```

This allows you to alter a data object in the calling function.

The compiler automatically generates the inline function definition.

No more forgetting &s.

### Function Return Values

The more long-winded `return` syntax in C++

- `return x;` becomes `return x;` in C++
- `return x + y;` becomes `return x + y;` in C++
- `return x * y;` becomes `return x * y;` in C++

#### Function Return Values

- `return` must be the last expression in the function.
- `return` syntax is optional.
- `return` syntax is preferred.

#### Type Casting

You must typecast a reference to an object of a different type.

```c
default struct { double re, im; } Complex;
```

is equivalent to

```c
typedef struct { double re, im; } Complex;
```

#### Variable Definition

Variable definitions may occur at the point at which they are first used.

```c
for (int J = 0; J < N; J++) ...
```

Variables definitions may occur at the point at which they are first used.

#### Type Precedence

You must specify the type of the object when you declare it.

```c
int x;
```

is equivalent to

```c
int x ;
```

#### Reference Type

References provide an alternate name (alias) for storage.

```c
typedef float (*p);  // p is a pointer to a float
```

### Miscellaneous Minor Extensions in C++
C++ Input and Output Example

```c++
#include <iostream>

int main()
{
    int val1, val2;
    std::cout << "Please enter two integers: " << std::endl;
    std::cin >> val1 >> val2;
    std::cout << "The sum of " << val1 << " and " << val2 << " is " << val1 + val2 << std::endl;
}
```

Note that `cout` >> endl writes new line and flushes output stream.

```c++
#include <iostream>

int main()
{
    int val1, val2;
    std::cout << "Please enter two integers: " << std::endl;
    std::cin >> val1 >> val2;
    std::cout << "The sum of " << val1 << " and " << val2 << " is " << val1 + val2 << std::endl;
}
```

Notethat `cout` << endl writes newline and flushes output stream.

NEW OPERATORS: `new` and `delete`

```c++
int *ptr1 = new int;
int *ptr2 = new int[50];
```

If successful, the first cell's address is stored in `ptr2`.

```c++
int *ptr1;
int *ptr2;
```

`delete` and `delete []` free storage allocated by `new`.

```c++
delete ptr1;
delete [] ptr2;
```

**WARNING:** The operators `new`, `delete`, and `delete []` should be used together and not intermixed with C storage management functions.

Function Overloading

```c++
void print(int i)
{
    printf("%d\n", i);
}

void print(char *s)
{
    printf("%s\n", s);
}
```

The function overloading can be overloaded except for `type`.

```c++
int main()
{
    (member selection (pointers))
    (member selection) (member selection)
    (member selection)
}
```

**Type:**

Parameter signature is used to resolve overloaded function at run-time.

```c++
int a, b;
```

**COMPILER COMPLIANCE BETWEEN `const` AND `non-const` VARIABLES:**

```c++
int a = 10;
const int b = 10;
```

**COMPILER REPORTS A WARNING IF THE SAME INDEX IS REFERENCED:**

```c++
const int a[10];
int b[10];
```

**C++ OPERATORS CAN BE OVERLOADED EXCEPT FOR `type`**

```c++
int main()
{
    :: (scope resolution) (member selection)
    (member selection)
    (member selection)
}
```

Function Overloading

```c++
int main()
{
    (member selection)
    (member selection)
    (member selection)
}
```