Evolution of Programming Languages

- **40's machine level**
  - raw binary
- **50's assembly language**
  - names for instructions and addresses
  - very specific to each machine
- **60's high-level languages: Fortran, Cobol, Algol, Basic**
- **70's system programming languages: C, PL/1, Algol 68, Pascal**
- **80's object-oriented languages: C++, Ada, Smalltalk, Objective C, …**

  strongly typed (to varying degrees)

  better control of large programs (at least in theory)

  better internal checks, organization, safety
- **90's scripting, Web, component-based, …: Perl, Java, Visual Basic, …**

  glue
- **00's Web server and client: Python, PHP, Ruby, Javascript, ...**

  focus on interfaces, components; frameworks

Program structure issues

- **how to cope with ever bigger programs?**
- **objects**
  - user-defined data types
- **components**
  - related objects
- **frameworks**
  - automatic generation of routine code
- **interfaces**
  - boundaries between code that provides a service and code that uses it
- **information hiding**
  - what parts of an implementation are visible
- **resource management**
  - creation and initialization of entities
  - maintaining state
  - ownership: sharing and copying
  - memory management
  - cleanup
- **error handling; exceptions**

Complicated data types in C

- **representation is visible, can't be protected**
  - opaque types are sort of an exception
- **creation and copying must be done very carefully**
  - and you don't get any help with them
- **no initialization**
  - you have to remember to do it
- **no help with deletion**
  - you have to recover the allocated memory when no longer in use
- **weak argument checking between declaration and call**
  - easy to get inconsistencies
- **the real problem: no abstraction mechanisms**
  - complicated data structures can be built, but access to the representation can't be controlled
  - you can't change your mind once the first implementation has been done
- **abstraction and information hiding are nice for small programs,absolutely necessary for big programs**

C++

- **designed & implemented by Bjarne Stroustrup**
  - began ~ 1980; ISO standard in 1998; still evolving (C++11 in Sept 2011)
- **a better C**
  - almost completely upwards compatible with C
  - more checking of interfaces (e.g., function prototypes, added to ANSI C)
  - other features for easier programming
- **data abstraction**
  - methods reveal only WHAT is done
  - classes hide HOW something is done in a program, can be changed as program evolves
- **object-oriented programming**
  - *inheritance* -- define new types that inherit properties from previous types
  - *polymorphism* or dynamic binding -- function to be called is determined by data type of specific object at run time
- **templates or "generic" programming**
  - compile-time parameterized types
  - define families of related types, where the type is a parameter
- **a "multi-paradigm" language**
  - lots of ways to write code

C++ classes

- **data abstraction and protection mechanism derived from Simula 67 (Kristen Nygaard, Norway)**

```
class Thing {
  public:
    methods -- functions for operations that can be done on this kind of object
  private:
    variables and functions that  implement the operations
};
```

- **defines a data type 'Thing'**
  - can declare variables and arrays of this type, create pointers to them, pass them to functions, return them, etc.
- **object: an instance of a class variable**
- **method: a function defined within the class**
- **private variables & functions not accessible from outside the class**
- **it is not possible to determine HOW the operations are implemented, only WHAT they do.**

C++ synopsis

- **data abstraction with classes**
  - a class defines a type that can be used to
    declare variables of that type, control access to representation
- **operator and function name overloading**
  - all C operators (including =, +=..., **( )**, **[ ]**, **->**, argument passing and function return but not . and ?:) can be overloaded to apply to user-defined types
- **control of creation and destruction of objects**
  - initialization of class objects, recovery of resources on destruction
- **inheritance: derived classes built on base classes**
  - virtual functions override base functions
  - multiple inheritance: inherit from more than one class
- **exception handling**
- **namespaces for separate libraries**
- **templates (generic types)**
  - Standard Template Library: generic algorithms on generic containers
  - template metaprogramming: execution of C++ code **during compilation**
- **compatible (almost) with C except for new keywords**

Topics

- **basics**
- **memory management, new/delete**
- **operator overloading**
- **references**
  - controlled behind-the-scenes pointers
- **constructors, destructors, assignment**
  - control of creation, copying and deletion of objects
- **inheritance**
  - class hierarchies
  - dynamic types (polymorphism)
- **templates**
  - compile-time parameterized types
- **Standard Template Library**
  - container classes, generic algorithms, iterators, function objects
- **performance**

Stack class in C++

```
// stk1.c:   simple-minded stack class
class stack {
   private:            // default visibility
        int stk[100];
        int *sp;
   public:
        int push(int);
        int pop();
        stack();     // constructor decl
};

int stack::push(int n) {
        return *sp++ = n;
}
int stack::pop() {
        return *--sp;
}
stack::stack() { // constructor implementation
       sp = stk;
}

stack s1, s2;   // calls constructors
s1.push(1);     // method calls
s2.push(s1.pop());
```

Inline definitions

- **member function body can be written inside the class definition**
- **this normally causes it to be implemented inline**
  - no function call overhead

```
// stk2.c:   inline member functions

class stack {
    int stk[100];
    int *sp;
  public:
    int push(int n)  { return *sp++ = n; }
    int pop()        { return *--sp; }
    stack()          { sp = stk; }
};
```

Memory allocation: new and delete

- **`new` is a type-safe alternative to malloc**
  - `delete` is the matching alternative to free
- **`new T` allocates an object of type T, returns pointer to it**

```
stack *sp = new stack;
```

- **`new T[n]` allocates array of T's, returns pointer to first**

```
int *stk = new int[100];
```

  - by default, throws exception if no memory
- **`delete p` frees the single item pointed to by p**

```
delete sp;
```

- **`delete [] p` frees the array beginning at p**

```
delete [] stk;
```

- **`new` uses T's constructor for objects of type T**
  - need a default constructor for array allocation
- **`delete` uses T's destructor `~T()`**
- **use `new/delete` instead of `malloc/free`**
  - malloc/free provide raw memory but no semantics
  - this is inadequate for objects with state
  - **never** mix new/delete and malloc/free

Dynamic stack with new, delete

```
// stk3.c: new, destructors, delete

class stack {
  private:
        int *stk;     // allocated dynamically
        int *sp;      // next free place
  public:
        int push(int);
        int pop();
        stack();      // constructor
        stack(int n); // constructor
        ~stack();     // destructor
};

stack::stack() {
        stk = new int[100];  sp = stk;
}
stack::stack(int n) {
        stk = new int[n];  sp = stk;
}
stack::~stack() {
        delete [ ] stk;
}
```

Constructors and destructors

- **constructor:**

  creating a new object (including initialization)
  - implicitly, by entering the scope where it is declared
  - explicitly, by calling <u>**`new`**</u>

- **destructor:**

  destroying an existing object (including cleanup)
  - implicitly, by leaving the scope where it is declared
  - explicitly, by calling <u>**`delete`**</u> on an object created by **`new`**

- **construction includes initialization, so it may be parameterized**
  - by multiple constructor functions with different args
  - an example of function overloading

- **`new` can be used to create an array of objects**
  - in which case **`delete`** can delete the entire array

Implicit and explicit allocation and deallocation

- **implicit:**

```
f() {
    int i;
    stack s;      // calls constructor stack::stack()
    ...
    // calls s.~stack() implicitly
}
```

- **explicit:**

```
f() {
    int *ip = new int;
    stack *sp = new stack;    // calls stack::stack()
    ...
    delete sp; // calls sp->~stack()
    delete ip;
    ...
}
```

Constructors; overloaded functions

- **two or more functions can have the same name if the number and/or types of arguments are different**

```
abs(int);   abs(double);   abs(complex)
atan(double x);    atan(double y, double x);

int abs(int x) { return x >= 0 ? x : -x; }
double abs(double x) { return x >= 0 ? x : -x; }
…
```

- **multiple constructors for a class are a common instance**

```
stack::stack( );
stack::stack(int stacksize);

stack s;           // default stack::stack()
stack s1();                  // same
stack s2(100);     // stack::stack(100)
stack s3 = 100;    // also stack::stack(100)
```

Overloaded functions; default args

- **default arguments: syntactic sugar for a single function**

```
stack::stack(int n = 100);
```

- **declaration can be repeated if the same**

- **explicit size in call**

```
stack s(500);
```

- **omitted size uses default value**

```
stack s;
```

- **overloaded functions: different functions, distinguished by argument types**
- **these are two different functions:**

```
stack::stack(int n);
stack::stack();
```

Operator overloading

- **almost all C operators can be overloaded**
  - a new meaning can be defined when one operand of an operator is a user-defined (class) type
  - define **operator +** for object of type **T**

    ```
    T T::operator+(int n) {...}
    T T::operator+(double d) {...}
    ```
  - define regular **+** for object(s) of type **T**

    ```
    T operator +(T f, int n) {...}
    ```
  - can't redefine operators for built-in types

    `int operator +(int, int)` is ILLEGAL
  - can't define new operators
  - can't change precedence and associativity

    e.g., ^ is low precedence even if used for exponentiation
- **3 short examples**
  - complex numbers: overloading arithmetic operators
  - IO streams: overloading << and >> for input and output
  - subscripting: overloading [ ]
- **later: overloading assignment and function calls**

Complex numbers

- **a complex number is a pair of doubles: (real part, imaginary part)**
- **supports arithmetic operations like +, -**
- **an arithmetic type for which operator overloading makes sense**
  - `complex` added as explicit type in 1999 C standard
  - in C++, can create it as needed
    use extension mechanism instead of extending language

- **also illustrates...**

- `friend` **declaration**
  - mechanism for controlled exposure of representation
  - classes can share representation
- **default constructors**
  - use of default arguments to simplify declarations
- **implicit coercions**
  - generalization of C promotion rules, based on constructors

An implementation of complex class

```
class complex {
    double re, im;
  public:
    complex(double r = 0, double i = 0)
        { re = r; im = i; }  // constructor

    friend complex operator +(complex,complex);
    friend complex operator *(complex,complex);
};

complex operator +(complex c1, complex c2) {
    return complex(c1.re+c2.re, c1.im+c2.im);
}
```

- **complex declarations and expressions**

```
complex a(1.1, 2.2), b(3.3), c(4), d;

d = 2 * a;
```

  2 coerced to 2.0 (*C* promotion rule)

  then constructor invoked to make complex(2.0, 0.0)

- **operator overloading works well for arithmetic types**

References: controlled pointers

- **need a way to access object, not a copy of it**
- **in *C*, use pointers**

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

- **in *C++*, references attach a name to an object**
- **a way to get "call by reference" (<u>var</u>) parameters without using explicit pointers**

```
void swap(int &x, int &y) {
   int temp;
   temp = x; x = y; y = temp;
}
swap(a, b);   // pointers are implicit
```

- **because it's really a pointer, a reference provides a way to access an object without copying it**

A vector class: overloading [ ]

```
class ivec {  // vector of ints
   int *v;          // pointer to an array
   int size;        // number of elements
  public:
   ivec(int n) { v = new int[size = n]; }

   int& operator [](int n) {  // checked
      assert(n >= 0 && n < size);
      return v[n];
   }
};

   ivec iv(10);   // declaration
   iv[10] = 1;    // checked access on left side of =
```

- **operator[ ] returns a reference**
- **a reference gives access to the object so it can be changed**
- **necessary so we can use [ ] on left side of assignment**

Iostreams: overloading >> and <<

- **I/O of user-defined types without function-call syntax**

- **C printf and scanf can be used in C++**
  - no type checking
  - no mechanism for I/O of user-defined types

- **Java System.out.printf(arglist)**
  - does some type checking
  - basically just calls toString method for each item

- **Iostream library**
  - overloads << for output, >> for input
  - permits I/O of sequence of expressions
  - natural integration of I/O for user-defined types

    same syntax and semantics as for built-in types
  - type safety for built-in and user-defined types

Output with iostreams

- **overload operator << for output**
  - very low precedence
  - left-associative, so

```
cout << e1 << e2 << e3
```

  - is parsed as

```
(((cout << e1) << e2) << e3)
```

```
#include <iostream>
  ostream& operator<<(ostream& os, const complex& c) {
    os << "(" << c.real() << ", " << c.imag() << ")";
    return os;
  }
```

- **takes a reference to iostream and data item**
- **returns the reference so can use same iostream for next expression**
- **each item is converted into the proper type**
- **iostreams cin, cout, cerr already open**
  - corresponding to stdin, stdout, stderr

Input with iostreams

- **overload operator >> for input**
  - very low precedence
  - left-associative, so

    ```
    cin >> e1 >> e2 >> e3
    ```
  - is parsed as

    ```
    (((cin >> e1) >> e2) >> e3)
    ```

```
char name[100];
double val;

while (cin >> name >> val) {
    cout << name << " = "
         << val << "\n";
}
```

- **takes a reference to iostream and reference to data item**
- **returns the reference so can use same iostream for next expression**
- **each item is converted into the proper type**

  `cin >> name` calls `istream& operator >>(istream&, char*)`

Formatter in C++

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

const int maxlen = 60;
string line;
void addword(const string&);
void println();

main(int argc, char **argv) {
    string word;
    while (cin >> word)
        addword(word);
    println();
}
void addword(const string& w) {
    if (line.length() + w.length() > maxlen)
        println();
    if (line.length() > 0)
        line += " ";
    line += w;
}
void println() {
    if (line.length() > 0) {
        cout << line << endl;
        line = "";
    }
}
```

Summary of references

- **reference is in effect a very constrained pointer**
  - points to a specific object
  - can't be changed, though whatever it points to can certainly be changed
- **provides control of pointer operations for applications where addresses must be passed for access to an object**
  - e.g., a function that will change something in the caller
  - like swap(x, y)
- **provides notational convenience**
  - compiler takes care of all * and & properly
- **permits some non-intuitive operations like the overloading of `[]`**
  - `int &operator[]` permits use of `[]` on left side of assignment
  - `v[e]` means `v.operator[...](e)`