EDA031 C++ Programming

"We're not in Kansas anymore."
Dorothy

EDA031 C++ Programming – Course and slides

"If I have seen further it is by standing on the shoulders of Giants."
Isaac Newton

Course and course material developed and updated continuously for 20 years by Per Holm – now retired.

History of C++

C++	C + Simula (object orientation) + multiple inheritance + overloading + templates + exceptions. Invented by Bjarne Stroustrup.
1980	C with classes
1983	C++ (C with classes + virtual functions + overloading + more)
Later	New versions containing multiple inheritance, templates, exceptions
1998	ISO standard, defines standard library
2011	New ISO standard, C++11
2014	Cleanup of C++11

C++11, the New Standard $\boxed{\text{C++11}}$

Stroustrup: “C++11 feels like a new language.” C++11 has many features that makes the language easier to learn and to use.

Work on a new C++ standard started early. Initially the new standard was called C++0X, and it was hoped that “X would be a decimal digit.” That didn’t happen, but the standard was finally accepted in 2011.

These slides (and Lippman’s book) describe the new standard. Additions to the base language and to the library are marked $\boxed{\texttt{C++11}}$.

Hello, world . . .

```
#include <iostream>  // input-output library

int main() {
    std::cout << "Hello, world" << std::endl;
}
```

- `main()` is a “free” function, not a method in a class
- Can have arguments similar to `String[] args` in Java
- The return type is `int`, returns the status code. `0` success, `!= 0` failure (`return 0` may be omitted in `main`, but not in other functions)
- `std::` is used to fetch names from the `std` “namespace” (similar to the Java package `java.lang`)

Compile, Link and Run

The Hello world program is in a file *hello.cc*.

```
g++ -c hello.cc       // compile, hello.cc -> hello.o
g++ -o hello hello.o // link, hello.o + library -> hello
./hello               // execute
```

- Usually, there are several more options on the command lines
- Usually, an executable consists of several `.o` files
- g++ is the GNU C++ compiler
- `clang++` is another compiler, from the LLVM project

Parameters on the Command Line

You can access parameters given on the command line if you define the main function like this:

```
int main(int argc, char* argv[]) { ... }
```

`argc` is the number of parameters, including the program name. `argv` is an array of C-style strings that contains the parameters (more about strings later). Example:

```
./myprog -a myfile.txt
    argc = 3
    argv[0] = "./myprog"
    argv[1] = "-a"
    argv[2] = "myfile.txt"
    argv[3] = 0             // end mark
```

Reading and Writing

```
#include <iostream>

int main() {
    int sum = 0;
    int value;
    while (std::cin >> value) {
        sum += value;
    }
    std::cout << "The sum is: " << sum << std::endl;
}
```

- `cin` and `cout` are input and output streams (stdin and stdout).
- `>>` is the input operator, reads whitespace-separated tokens. Returns `false` on end-of-file.
- `<<` is the output operator.

Reading and Writing Files

You can create stream objects in order to read from or write to files (`ifstream` for reading, `ofstream` for writing). Read from a file whose name is given as the first argument on the command line:

```
#include <fstream>  // file streams
#include <iostream>

int main(int argc, char* argv[]) {
    std::ifstream input(argv[1]);
    if (!input) { // "if the file couldn't be opened"
        std::cerr << "Could not open: " << argv[1] << endl;
        exit(1);
    }
    // ... read with input >> ...
    input.close();
}
```

In C++11, the filename can also be a `string` object. `C++11`

A Class Definition

```
class Point {
public:
    Point(double ix, double iy); // constructor
    double getX() const;         // const: does not modify
    double getY() const;         // the state of the object
    void move(double dx, double dy); // does modify, not const
private:
    double x;                    // attributes
    double y;
};                               // note semicolon
```

- Methods are called member functions in C++, attributes are called member variables
- Member functions are only declared here, the definitions are in a separate file
- `public` and `private` sections, arbitrary order, `private` is default

Definition of Member Functions

```
Point::Point(double ix, double iy) {
    x = ix; // this is ok but there are better ways to
    y = iy; // initialize member variables, see next slide
}

double Point::getX() const {
    return x;
}

double Point::getY() const {
    return y;
}

void Point::move(double dx, double dy) {
    x += dx;
    y += dy;
}
```

- `Point::` indicates that the functions belong to class `Point`

Initialization of Member Variables

Normally, member variables are not initialized using assignments in the constructor body. Instead, they are initialized in the constructor *initializer list*:

```
Point::Point(double ix, double iy) : x(ix), y(iy) {}
```

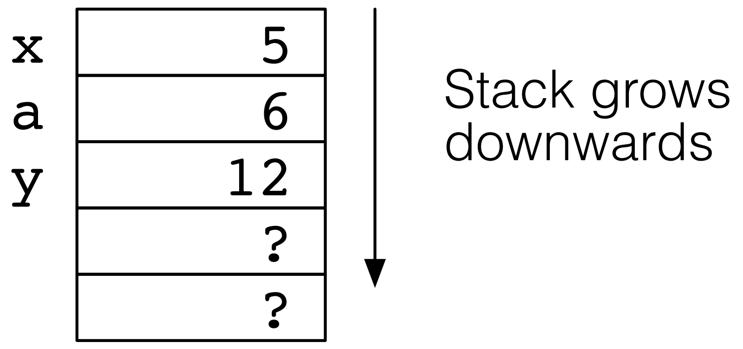
- Syntax: `member-variable(initializer-expression)`
- It is usually better to use the initializer list, and sometimes necessary. Discussed further later.

Stack, Heap, Activation Records

The (virtual) memory for a program is divided into two areas: the *stack* for local variables in functions, the *heap* for dynamically allocated variables.

When a function is entered, memory is allocated on the stack for parameters and local variables. This memory area is called an *activation record* or *stack frame*. When the function is exited, the activation record is deallocated.

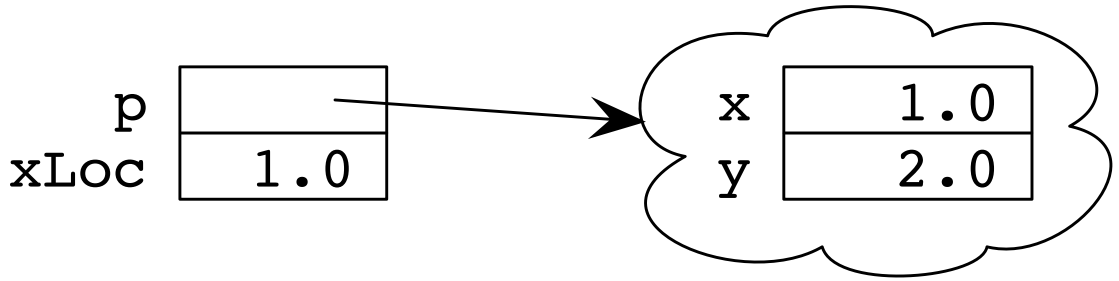
```
int main() {
    int x = 5;
    f(x + 1);
}

void f(int a) {
    int y = 2 * a;
}
```



Objects in Java

A Java object is always allocated on the heap, and you need a pointer (reference variable) to keep track of the object:

```
void f() {
    Point p = new Point(1, 2);
    double xLoc = p.getX();
    ...
}
```



When a Java object can no longer be reached (no references point to the object) it becomes garbage. The garbage collector reclaims memory used by unreachable objects.

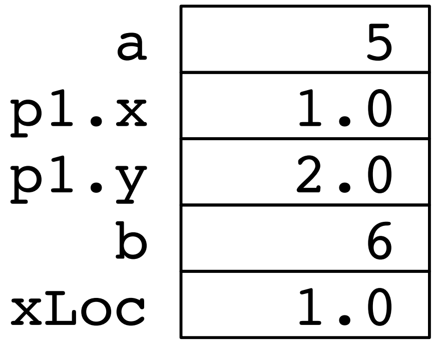
Objects in C++, Stack Allocation

In C++, objects can be allocated in several different ways. The two most common are:

- stack allocated (automatic) — like any local variable, accessed by name
- heap allocated — like in Java, accessed through pointer

Stack allocation example:

```
void f() {
    int a = 5;
    Point p1(1, 2);
    int b = 6;
    double xLoc = p1.getX();
    ...
}
```

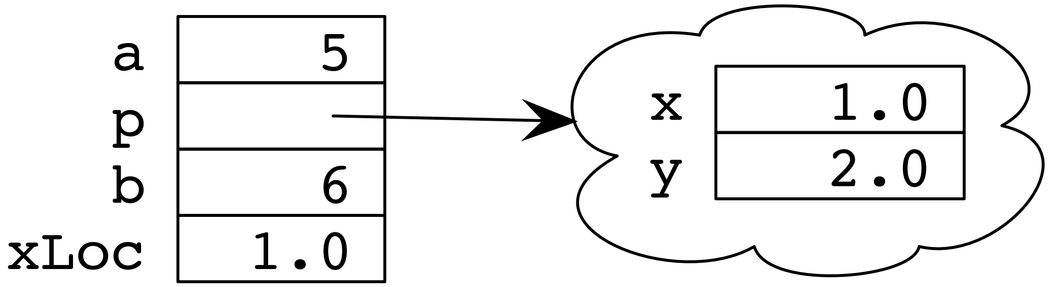


p1 (and a, b, and xLoc) are deallocated automatically when f is exited.

Heap Allocation

An object on the heap is similar to a Java object, but in C++ there is no garbage collector so you must explicitly delete the object.

```
void f() {
    int a = 5;
    Point* p = new Point(1, 2);
    int b = 6;
    double xLoc = p->getX();
    ...
    delete p;
}
```



`Point* p` means that `p` is a pointer, `p->` is used to access members in the object. The lifetime of a heap-allocated object is between `new` and `delete`; it has nothing to do with the block structure.

If you forget to delete an object, the result is a *memory leak*.

Safe Pointers C++11

In C++11, there are new library classes that encapsulate “raw” pointers and make sure that dynamic objects always are deallocated. The two most important:

`unique_ptr`	A pointer which “owns” its object. If a pointer is reassigned, ownership is transferred.
`shared_ptr`	A pointer which counts the number of references to the object. The object is deleted when it has no more references.

Do Not Use Dynamic Memory

Java programmers that come to C++ often continue programming in the Java style. This means, for example, that they create all objects with `new Classname`. This is seldom necessary.

It is only necessary to allocate an object on the heap (with `new`) if the object shall be shared, if the size of the object is unknown, or if the object must outlive the current scope. Examples:

- Implement a dynamic vector (like a Java `ArrayList`), or a dynamic string (like a Java `String` or `StringBuilder`), or . . . There are library classes for such things.
- Implement a linked list (where the nodes are created in a member function), or . . . There are library classes for such things.
- Handle polymorphic objects, handle objects that must be shared, . . .