CSCI 2270: Data Structures

Lecture 04: C++ Review: Functions, Recursion, Stack versus Heap Memory

Ashutosh Trivedi

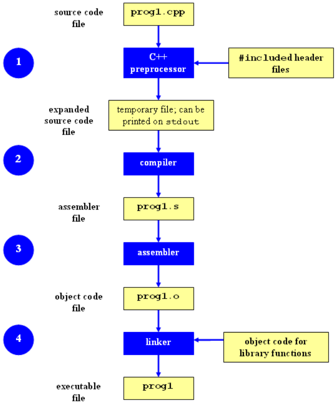
Department of Computer Science
UNIVERSITY OF COLORADO BOULDER

C++: A quick review (contd.)

C++ Compilation process

Compiling a source code file in C++ is a four-step process.

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

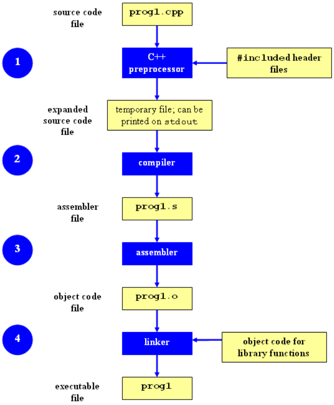


– The preprocessor copies the included header files into the source code, generates macro code, and replaces symbolic constants (`#define`-s).

C++ Compilation process

Compiling a source code file in C++ is a four-step process.

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

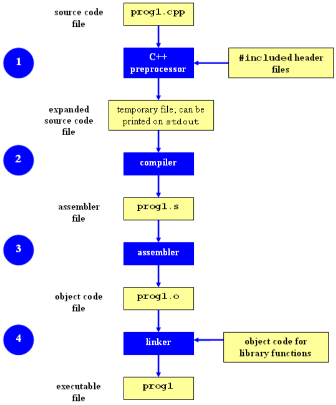


- The preprocessor copies the included header files into the source code, generates macro code, and replaces symbolic constants (`#define`-s).
- The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.

C++ Compilation process

Compiling a source code file in C++ is a four-step process.

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

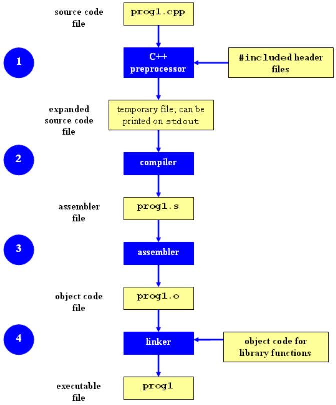


- The preprocessor copies the included header files into the source code, generates macro code, and replaces symbolic constants (`#define`-s).
- The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
- The assembler code generated by the compiler is assembled into the object code for the platform.

C++ Compilation process

Compiling a source code file in C++ is a four-step process.

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```

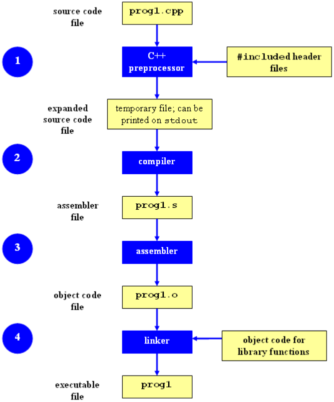


- The preprocessor copies the included header files into the source code, generates macro code, and replaces symbolic constants (`#define`-s).
- The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
- The assembler code generated by the compiler is assembled into the object code for the platform.
- The object code generated by the assembler is linked with the library functions to produce an executable file.

C++ Compilation process

Compiling a source code file in C++ is a four-step process.

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```



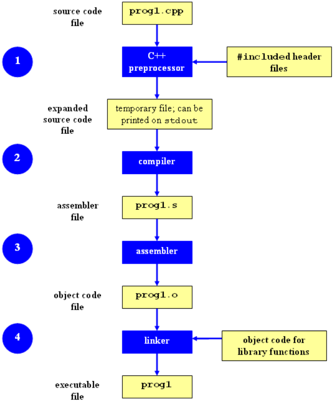
– To pause the process after the preprocessor step:

```
g++ -Wall -std=c++11 -E prog1.cpp
```

C++ Compilation process

Compiling a source code file in C++ is a four-step process.

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```



- To pause the process after the preprocessor step:

```
g++ -Wall -std=c++11 -E prog1.cpp
```

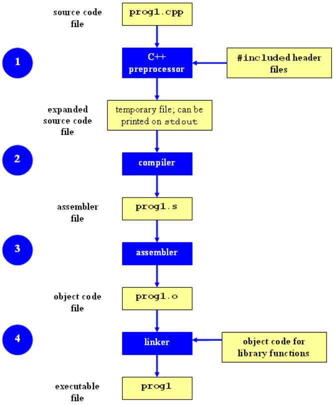
- To pause after the compiler step:

```
g++ -Wall -std=c++11 -S prog1.cpp
```

C++ Compilation process

Compiling a source code file in C++ is a four-step process.

```
g++ -Wall -std=c++11 -o prog1 prog1.cpp
```



- To pause the process after the preprocessor step:

```
g++ -Wall -std=c++11 -E prog1.cpp
```

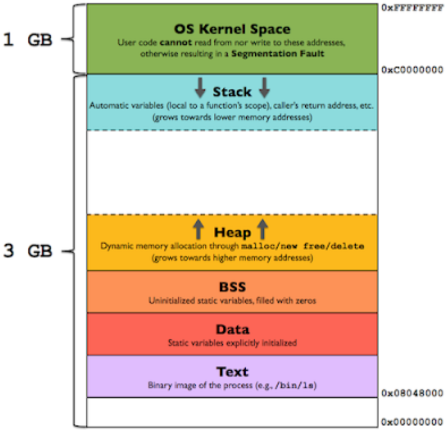
- To pause after the compiler step:

```
g++ -Wall -std=c++11 -S prog1.cpp
```

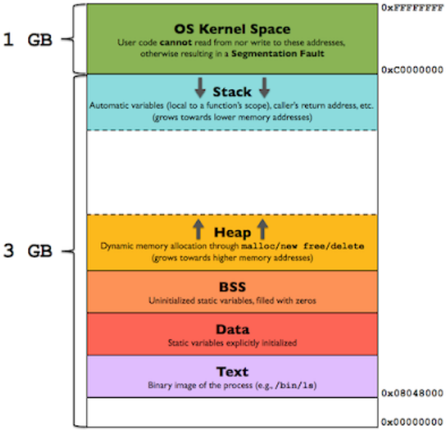
- To pause after the assembler step:

```
g++ -Wall -std=c++11 -c prog1.cpp
```

In-Memory Layout of a Program



In-Memory Layout of a Program



– Segmentation fault: *the program has attempted to access a restricted area of memory (memory access violation)*

– core dump: *dump of the process state before segmentation fault*

Functions: Declaration and Definition

```cpp
// program21.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3)  std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  while (y-->1) z = z*x;
  return z;
}
```

Functions: Declaration and Definition

What is wrong with the following program?

```
// program22.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3) {
    std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
    std::cout << w;
    std::cout << power(std::stoi(argv[1]));
  }
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  int w = x*y;
  while (y-->1) z = z*x;
  return z;
}
```

Functions: Declaration and Definition

What is wrong with the following program?

```
// program22.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3) {
    std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
    std::cout << w;
    std::cout << power(std::stoi(argv[1]));
  }
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  int w = x*y;
  while (y-->1) z = z*x;
  return z;
}
```

– Variable w is not accessible inside the function `main`.

Functions: Declaration and Definition

What is wrong with the following program?

```
// program22.cpp
#include<iostream>
#include <cmath>
int square(int x);
long exponentiation(int x, int y);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << square(std::stoi(argv[1]));
  if (argc == 3) {
    std::cout << exponentiation(std::stoi(argv[1]), std::stoi(argv[2]));
    std::cout << w;
    std::cout << power(std::stoi(argv[1]));
  }
  return 0;
}
int square(int x) {
  int y = x*x;
  return y;
}
long exponentiation(int x, int y) {
  long z = x;
  int w = x*y;
  while (y-->1) z = z*x;
  return z;
}
```

- Variable w is not accessible inside the function `main`.
- Notice that the variable `argv` is still available to `main` after the exponentiation procedure returns. Why?

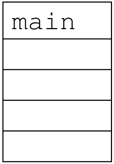
Functions: Factorial

```
// program23.cpp
// Factorial(n) = n * (n-1) * (n-2) * ... * 1
#include<iostream>
#include <cmath>
int factorial(int x);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << factorial(std::stoi(argv[1]));
  return 0;
}
int factorial(int x) {
  int res = x;
  while (x-- > 1) res = res * x;
  return res;
}
```

Functions: Factorial (recursive)

```
// program24.cpp
// Factorial(n) = n * Factorial(n-1)
#include<iostream>
#include <cmath>
int factorial(int x);
int main(int argc, char* argv[])  {
  if (argc == 2)  std::cout << factorial(std::stoi(argv[1]));
  return 0;
}
int factorial(int x) {
  int y = 1;
  std::cout << "Entering function Factorial(" << x << ")" << std::endl;
  if (x == 0) y = 1;
  else y =  x * factorial(x-1);
  std::cout << "Computed  Factorial(" << x << ")" << std::endl;
  return y;
}
```

Function



Function

main
`factorial(3) = 3* factorial(2)`

Function

`main`
`factorial(3) = 3* factorial(2)`
`factorial(2) = 2* factorial(1)`

Function

main
factorial(3) = 3* factorial(2)
factorial(2) = 2* factorial(1)
factorial(1) = 1* factorial(0)

Function

main
factorial(3) = 3* factorial(2)
factorial(2) = 2* factorial(1)
factorial(1) = 1* factorial(0)
factorial(0) = 1

Function

main
factorial(3) = 3* factorial(2)
factorial(2) = 2* factorial(1)
factorial(1) = 1* factorial(0)

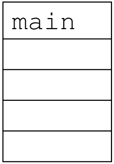
Function

main
factorial(3) = 3* factorial(2)
factorial(2) = 2* factorial(1)

Function

main
`factorial(3) = 3* factorial(2)`

Function



Stack Memory

- A “stack frame” is created when a function is called. Henceforth, all the local variables of that function are created within the confines of this stack frame.
- When the function returns, its stack frame is “deleted”. The deletion of all variables happens “automagically.’
- Programmer does not need to concern herself to create or to delete copy of variables. The run-time system provides this facility.

Stack Memory

- A “stack frame” is created when a function is called. Henceforth, all the local variables of that function are created within the confines of this stack frame.
- When the function returns, its stack frame is “deleted”. The deletion of all variables happens “automagically.’
- Programmer does not need to concern herself to create or to delete copy of variables. The run-time system provides this facility.
- A big limitation of stack: how to store variables that one can access across function calls?

Stack Memory

- A “stack frame” is created when a function is called. Henceforth, all the local variables of that function are created within the confines of this stack frame.
- When the function returns, its stack frame is “deleted”. The deletion of all variables happens “automagically.’
- Programmer does not need to concern herself to create or to delete copy of variables. The run-time system provides this facility.
- A big limitation of stack: how to store variables that one can access across function calls?
- Such a memory to store global variables is called the heap memory.

Stack Memory: summary

1. The stack grows and shrinks as functions push and pop local variables.
2. There is no need to manage the memory yourself, variables are allocated and freed automagically.
3. The stack has size limits. (Check yours with `ulimit -a` and set with `ulimit -s 33333`.
4. The stack variables only exist while the function that created them, is running.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).
- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

The Heap

- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU.
- To allocate memory on the heap, you must use operator `new`.
- Once you have allocated memory on the heap, you are responsible for deleting it using `delete` operator to deallocate that memory once you don't need it any more.
- If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes).
- There are debuggers (valgrind) that can help you detect memory leaks.
- Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer).
- Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.
- Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap. We will talk about pointers in the next lecture.

Stack Vs Heap: Pros and Cons

Stack:

- very fast access
- don't have to explicitly free variables
- space is managed efficiently by CPU,
- memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)
- variables cannot be resized

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.
- If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack.

When to use the Heap?

- If you need to allocate a large block of memory (e.g. a large array, or a big struct), and you need to keep that variable around a long time (like a global), then you should allocate it on the heap.
- If you are dealing with relatively small variables that only need to persist as long as the function using them is alive, then you should use the stack.
- If you need variables like arrays and structs that can change size dynamically (e.g. arrays that can grow or shrink as needed) then you will likely need to allocate them on the heap, and use dynamic memory allocation functions to manage that memory "by hand".

Pointer Variables

```
// program7.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  char ch= 'a';
  char *cp; // cp is a pointer variable
  cp = &ch; // cp points to the address of the ch
  std::cout << "Size of a pointer to char: ";
  std::cout << sizeof(char *) << std::endl;
  std::cout << "Address of ch is = " << (void *) cp;
  return 0;
}
```

1. What are the sizes of pointers to different types of objects?

Pointer Variables

```
// program7.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  char ch= 'a';
  char *cp; // cp is a pointer variable
  cp = &ch; // cp points to the address of the ch
  std::cout << "Size of a pointer to char: ";
  std::cout << sizeof(char *) << std::endl;
  std::cout << "Address of ch is = " << (void *) cp;
  return 0;
}
```

1. What are the sizes of pointers to different types of objects?
2. Repeat the above exercise for other types.

Pointer Variables

```
// program7.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  char ch= 'a';
  char *cp; // cp is a pointer variable
  cp = &ch; // cp points to the address of the ch
  std::cout << "Size of a pointer to char: ";
  std::cout << sizeof(char *) << std::endl;
  std::cout << "Address of ch is = " << (void *) cp;
  return 0;
}
```

1. What are the sizes of pointers to different types of objects?
2. Repeat the above exercise for other types.
3. A pointer to variable of type T is:
   3.1 `T* p` or `T *p`
   3.2 bad practice: `int *p, q, r`.

Pointer Variables

```
// program7.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  char ch= 'a';
  char *cp; // cp is a pointer variable
  cp = &ch; // cp points to the address of the ch
  std::cout << "Size of a pointer to char: ";
  std::cout << sizeof(char *) << std::endl;
  std::cout << "Address of ch is = " << (void *) cp;
  return 0;
}
```

1. What are the sizes of pointers to different types of objects?
2. Repeat the above exercise for other types.
3. A pointer to variable of type T is:
   3.1 `T* p` or `T *p`
   3.2 bad practice: `int *p, q, r`.
4. A pointer variable equal to 0 means it does not refer to an object. Use of **NULL** discouraged!

Arrays

- An array is a collection of elements of the same type.
- Given a variable of type T, and array of type $T[N]$ holds an array of N elements, each of type N.
- Each element of the array can be referenced by its index that is a number of 0 to $N-1$.

Arrays

- An array is a collection of elements of the same type.
- Given a variable of type T, and array of type $T[N]$ holds an array of N elements, each of type N.
- Each element of the array can be referenced by its index that is a number of 0 to $N - 1$.

```
// program8.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int ia[3]; //Array of 3 ints with garbage values
  std::cout << ia[1] << std::endl;
  float fa[] = {1, 2, 3}; //Array of 3 floats initialzed: size automatically computed
  std::cout << fa[2] << std::endl; // Read different values
  return 0;
}
```

Arrays (Statically Declared Arrays)

```
// program8.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int ia[3]; //Array of 3 ints with garbage values
  std::cout << ia[1] << std::endl;
  float fa[] = {1, 2, 3}; //Array of 3 floats initialzed: size automatically computed
  std::cout << fa[2] << std::endl; // Read different values
  return 0;
}
```

1. Static Array storage is contiguous.
2. Array bound must be a constant expression. If you need variable bounds, use a `vector`.
3. What happens when initialization and array size mismatch?
4. Multi-dimensional arrays (contiguous in row-order fashion!).

Arrays (Dynamically Declared Arrays)

```
// program9.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int* pa = 0; // pa is a pointer to integers
  int n;
  std::cout <<  "Enter dynamically allocated array size:";
  std::cin >> n;
  pa = new int[n];
  for (int i = 0; i < n; i++) {
    pa[i] = i;
  }
  // Use a as a normal array
  delete[] pa; // When done, free memory pointed to by a.
  pa = 0; //// Clear a to prevent using invalid memory reference.
  return 0;
}
```

References (A Rose by another name!)

```
// program10.cpp
#include<iostream>
int main(int argc, char* argv[])
{
  int i = 1;
  int &r = i; // r and i refer to same int
  // int &s;  // Error: must be initialized unless "extern"
  int x = r;  // x = 1
  r = 2;      // x = 2
  return 0;
}
```