Chapter 8. 인라인의 기본

**인라인 이란 무엇인가?**

- 어떤 메소드에서 다른 메소드를 호출할 때, 인라인은 메소드 호출을 매크로와 유사하게 확장

- 인라인 여부를 결정하는 두가지 방법이 있다.

1. 메소드 정의에 예약된 단어 inline을 지정하는 것,

2. 선언 헤더에서 메소드를 정의하는 것

|  |
| --- |
| class GatedInt  {  public:  int get()  {  return n;  }  void set(int x) { n = x; }  private:  int n;  }; |

- get,set 함수는 선언 내에 정의 되었기 때문에 둘 다 인라인

- 반대로, 클래스 선언 외부에 정의하면서 헤더 파일 내부 혹은 헤더 파일에 포함된 파일 내부에 정의한 메소드도 인라인

|  |
| --- |
| class GatedInt  {  public:  int get();  void set(int x);  private:  int n;  };  inline int GatedInt::get() { return n; }  inline void GatedInt::set(int x) { n = x; } |

- 인라인은 일반 메소드 처럼 컴파일 되지 않는다.

- 인라인 메소드의 정의가 자신의 호출 메소드에 포함되는 사실은 인라인 메소드를 수정하면 이 메소드를 사용하는 몯느 모듈을 완전히 새로 컴파일 해야 함을 의미, **성능 개선을 누리기 위한 컴파일 시간의 증가**이다.

- 인라인 메소드를 프로그램에서 다음과 같이 호출

|  |
| --- |
| int main()  {  GatedInt gi;  gi.set(12);  std::cout << gi.get();  return 0;  } |

- inline get과 set을 사용하면 main 프로그램에서 함수는 논리적으로 볼 때 다음과 같다.

|  |
| --- |
| int main()  {  GatedInt gi;  {  gi.n = 12;  }  int temp = gi.get();  std::cout << temp;  return 0;  } |

- 다음 컴파일러의 최적화 작업을 수행한다면, main에 대한 최적화 작업의 결과는 다음과 같다.

|  |
| --- |
| int main()  {  std::cout << 12;  return 0;  } |

- 컴파일러가 메소드를 인라인으로 만들기 위해 사용하는 방법

1. 인라인을오 만들 메소드의 연속된 코드 블록을 이 메소드를 호출하는 메소드의 호출 지점에 복사

2. 인라인 메소드의 지역 변수들을 블록에 할당, 인라인 메소드의 입력 인자와 반환값을 호출 메소드 지역 변수 공간에 매핑

3. 만약 인라인 메소드가 여러 반환 지점을 가지고 있다면, 이 반환 지점들은 인라인 블록의 마지막으로 분기될 수 있음.

4. 함수 호출과 관련된 모든 자취를 없애고, 함수 호출에 관한 인과 성능 부하를 없앤다.

- 하지만 호출 부하를 없애는 것은 인라인에 관한 성능 이야기의 절반 뿐이다.

|  |
| --- |
| int x::y(int a)  {  int b = 6;  // b는 이 섹션에서 수정 되지 않음  int m = build\_mask(b);  // m은 이 섹션에서 수정 되지 않는다.  int n = m + 1;  }  inline int build\_mask(int q)  {  if (q > 65535) return -1;  else if (q > 0) return (1 << q) - 1;  else return 0;  } |

- y 내부에서 build\_mask를 인라인으로 만들면서 최적화를 수행하지 않으면 다음과 같은 결과가 발생

|  |
| --- |
| int x::y(int a)  {  int b = 6;  // b는 이 섹션에서 수정 되지 않음  int m;  {  int \_temp\_q = 6;  int \_temp;  if (\_temp\_q > 65535) \_temp = -1;  else if (\_temp\_q > 0) \_temp = (1 << \_temp\_q) - 1;  else \_temp = 0;  m = \_temp;  }  // m은 이 섹션에서 수정 되지 않는다.  int n = m + 1;  } |

- 하지만 최적화된 결과는 다음과 같다.

|  |
| --- |
| int x::y(int a)  {  int b = 6;  // b는 이 섹션에서 수정 되지 않음  int m = 0x3F;  // m은 이 섹션에서 수정 되지 않는다.  int n = 0x40;  } |

- 호출간 최적화는 인라인 최적화 방정식의 나머지 절반을 차지, 최적화를 잘 수행하는 컴파일러는 인라인 메소드의 블록 경계를 인식하지 못하도록 만듬, 어떤 경우에 메소드의 전체 코드가 최적화 되어 없어질 수도 있다. 컴파일러는 많은 메소드의 순서를 다시 정렬할 수 있다.

- 그러므로 논리적으로 볼 때 인라인 메소드를 어떤 결속성을 유지하는 것으로 생각하면 편리하겠지만, 반드시 이러한 경우에만 일어나는 것은 아니며, 이것이 인라인의 주요 이점이다.

**메소드 호출 부하**

- 인라인이 주는 이점을 이해 하기 위해 **메소드 호출과 반환**에 어떤 사항들이 포함되어 있는지 이해해야 할 것

- 시스템은 세 개 혹은 네 개의 “하우스 키핑(housekeeping)”레지스터를 보유, 명령 포인터(Instruction Pointer, 이 레지스터는 프로그램을 세지 않지만 프로그램 카운터 라고 불리기도 한다.), 링크 레지스터(Link Register), 스택 포인터(Stack, Pointer), 프레임 포인터(Frame Pointer), 인자 포인터(Argument Pointer) 약자로, 각각 **IP, LR, SP, FP, AP**이다.

각 레지스터의 기능

**- 명령 포인터 (IP):** **다음 실행할 명령의 주소를 저장**, 메소드를 호출하면 호출된 메소드의 명령으로 점프해야 하기 떄문에 IP를 수정해야 한다. 하지만 IP를 단순히 덮어 쓸 수는 없으며, IP의 값이 변하기 전에 이전 값을 저장해 놓아야 한다. 만약 그렇지 않다면 호출 메소드로 다시 돌아갈 방법이 없어진다.

**- 링크 레지스터 (LR) : 현재 메소드를 호출한 메소드의 주소 IP를 포함하고 있다.** 이것은 메소드가 자신의 실행을 완료한 다음 어느 곳으로 돌아갈 지에 관한 위치이다. LR은 전형적으로 아키텍처의 호출 명령 연산과 밀접하며, LR의 값은 함수를 호출함과 동시에 부가적으로 설정된다.

**- 스택 포인터 (SP) : 얼마나 많은 스택이 점유 되었는가에 관한 정보를 저장하고 있다.** 메소드를 호출하면 스택 공간을 점유하게 되고, 메소드가 반환하면 이전에 할당된 스택 공간을 해지한다. 호출자의 IP와 LR과 비슷하게, 메소드가 반환하고 나면 스택에 전달되었던 인자들을 모두 조정하여 스택을 복구스켜야 한다.

**- 인자 포인터(AP), 프레임 포인터 (FP) :** 시스템 의존적, 어떤 아키텍처는 둘 다 가지고 있지 않으며, 오직 하나만 가진 시스템, 둘 다 가진 시스템도 존재**, FP를 사용하여 스택의 두 가지 영역 사이의 경계를 표시, 하나의 영역에는 유지되어야 할 상태를 가진 레지스터의 값을 호출 메소드가 저장하고, 다른 영역에는 호출된 메소드의 자동 변수들이 저장된다**. **AP는 메소드에 전달되는 인자가 스택 어디에 있는지 가리키고 있어야 한다.**

**왜 인라인 인가?**

- 인라인은 어떠한 것도 새로 작성하지 않고, 상대적으로 큰 시스템의 성능을 빠르게 변환할 수 있다.

**키 포인트**

- 인라인은 메소드 호출을 메소드의 코드로 교체하는 작업이다.

- 인라인은 호출 오버헤드를 없애고 호출간 최적화가 일어날 수 있도록 하여 성능을 향상 시킨다.

- 인라인은 주로 실행 시간 최적화 이지만, 실행 이미지의 크기도 작게 만들어 준다.